Multilist Scheduling:
A New Parallel Programming Model

I-Chen Wu

July 30, 1993
CMU-CS-93-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:
H.T. Kung, Chair
Peter Steenkiste
David O'Hallaron
Gerald Thompson, GSIA
David Farber, University of Pennsylvania

This research was sponsored by the Advanced Research Projects Agency, Information Science and Technology Office, under the title *Research on Parallel Computing*, issued by ARPA/CMO under Contract MDA972-90-C-0035, ARPA Order No. 7330.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: parallel programming, task scheduling, multilist scheduling, dynamic load balancing, scheduling list, network-based multicomputer, divide-and-conquer, best-first search, network simulation, alpha-beta search, quicksort, factoring, set covering, range selection.

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Multilist Scheduling: A New Parallel Programming Model

I-CHEN WU

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

H. T. Kung — THESIS COMMITTEE CHAIR — August 1, 1993 — DATE

[signature] — DEPARTMENT HEAD — 8/12/93 — DATE

APPROVED:

R. Rashid — DEAN — 8/12/93 — DATE

Abstract

Parallel programming requires task scheduling to optimize performance; this primarily involves balancing the load over the processors. In many cases, it is critical to perform task scheduling at runtime. For example, (1) in many parallel applications the task load cannot be accurately predicted *a priori*; (2) in a network-based multicomputer the computational power of each processor may not remain constant. In order to support dynamic task scheduling, the programmer usually needs to design and implement a complex set of scheduling routines, e.g., routines for maintaining task lists and handling interprocessor communication for load balancing. Unfortunately, it is very difficult and time-consuming to write and debug all of these scheduling routines.

This thesis proposes a new approach which can greatly reduce the effort of developing efficient dynamic task scheduling routines. In our new approach, we decompose task scheduling into two parts — the specification of scheduling policies and the implementation of supportive scheduling operations — and then hide the latter from the programmer. We call this approach *multilist scheduling*, because it is based on a uniform scheduling model involving the use of multiple scheduling lists.

This thesis analyzes three main features of the new multilist scheduling model: ease of use, generality, and efficiency.

- Ease of use: Programmers only need to specify scheduling policies, not the details of supportive scheduling routines. Typically, this involves writing only tens of lines of C code, as opposed to thousands of lines of code for the supportive scheduling routines.
- Generality: We show that this model results in no loss of generality. We also illustrate the generality of the model by rendering several scheduling algorithms in the framework of our model, including the scheduling algorithms for parallel divide-and-conquer (D&C) and best-first search (BFS).
- Efficiency: We propose some efficient techniques for implementing scheduling lists, and also show that our general approach incurs no significant performance overhead, at least for the parallel D&C and BFS scheduling algorithms. In addition, we also demonstrate good performance results for some applications that are based on parallel D&C and BFS, such as the set covering problem.

Traditionally, it has been difficult to efficiently support both parallel D&C and BFS in a uniform framework. We believe that our system is the first system that do so.

Multilist scheduling is the first model which can hide the details of dynamic task scheduling routines while supporting general task scheduling. We expect that this model will have a significant impact on parallel programming, especially in the domain of multicomputers.

Acknowledgment

I would like to deeply thank all of those who helped me with this thesis. My advisor, H.T. Kung, gave me excellent advice and vision which led this thesis towards providing a *general* parallel programming system for dynamic task scheduling. My committee members, H.T. Kung, Peter Steenkiste, Dave O'Hallaron, Gerald Thompson, and David Farber, also offered valuable comments to improve the quality of the thesis. Especially, H.T. Kung, Peter Steenkiste, and Dave O'Hallaron also gave many useful comments in the early stages. Gerald Thompson generously provided his set covering program as an example application of my system. Hiroshi Nishikawa gave some valuable discussions about his language Aroma and its applications, which motivated the study of the scheduling algorithms in Sections 3.1.3 and 3.2.4. Joe Tebelskis carefully and patiently read several drafts of the thesis and helped me to improve the readability of this presentation. Susan Hinrichs, Bruce Siegell, and Jay Sipelstein also helped improve the presentation of this thesis. Peter Steenkiste, Bruce Siegell, Hiroshi Nishikawa, Michael Gillinov, and Ming-Jen Chan have helped me on some questions about operating systems and the Nectar system environment. In addition, during the early period of my Ph.D program when I was working on the Apply compiler, Jon Webb gave me some useful advice which was indirectly helpful for my thesis work.

I am grateful to many friends for their friendship and encouragement. Joe Tebelskis, in particular, an almost-6-year officemate of mine, has always generously provided help whenever I needed it and encouraged me whenever I had a difficult time. Never will I forget his genuine and enjoyable friendship.

I am indebted to my family for their support. My wife, Shwu-Huey, shared in my hard work with love, understanding, and encouragement. My son, Tony, and my daughter, Ariel, brought joy and happiness to me. Finally, I would like to dedicate this thesis to my parents, Wen-Ching and Kang-Bau, for their unconditional support and care in my life.

獻給我的父母：吳文慶和吳康抱

(To my parents, Wen-Ching and Kang-Bow)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel programming requires task scheduling to optimize performance; this primarily involves balancing the load over the processors. Task scheduling can be classified into *static task scheduling* or *dynamic task scheduling*, according to the time when task scheduling is performed. Static task scheduling is typically done by distributing the load of tasks evenly (or roughly evenly) over processors at the beginning of the job. This can be done with the help of compilers [39, 80, 100]. However, in many cases, it is critical to perform task scheduling at runtime:

- In many parallel applications, the task load cannot be accurately predicted *a priori*. For example, in mathematical optimization problems [35, 76], which usually involve large-scale tree searching, it is impossible to make useful *a priori* estimates on the size of the search tree or the sizes of its nodes.

- In *network-based multicomputers*[1], the computational power of each processor may not remain constant. For example, in a network-based multicomputer, someone may unexpectedly come to work on one of the participating workstations, slowing it down. Static task scheduling (or static load balancing) cannot balance the load well in such a situation.

[1] A network-based multicomputer is a number of computers (e.g., workstations), connected via a network, cooperating on the same job. As networks have grown more efficient, network-based multicomputers [61] have recently emerged as a new and attractive type of parallel system due to resource sharing which results in flexibility and low cost. We expect more and more applications to run on such multicomputers.

We will first describe the traditional approach to dynamic task scheduling (or dynamic load balancing) and point out the problem of the approach in Section 1.1. Then, we will briefly describe our new approach in Section 1.2. Finally, we will give an overview of this thesis in Section 1.3.

1.1 Traditional Approach

The traditional approach to dynamic scheduling is *ad hoc*: for each application or each class of applications, a task scheduling algorithm is implemented from scratch. For example:

- For a large class of applications which can be solved without concern for the scheduling sequence of executable tasks, many researchers [31, 67, 89, 107] have proposed various load balancing techniques to efficiently parallelize the applications.
- For tree search problems (e.g., *divide-and-conquer* (*D&C*) problems) which can be efficiently solved by using *depth-first search* (*DFS*), many researchers [32, 33, 34, 44, 82, 104, 105, 108] have proposed scheduling algorithms which can reduce the amount of communication while balancing the load.
- For tree search problems which can be efficiently solved by using *best-first search* (*BFS*), several researchers [1, 5, 46, 60, 76, 81, 101, 108] have proposed ways to parallelize BFS.
- For game tree search problems which can be solved by using alpha-beta (α-β) search, researchers have proposed many different kinds of parallel scheduling algorithms, e.g., the *mandatory work first* algorithm [4], the *principle variation splitting* (*PVS*) algorithm [21, 69], and other variations [11, 45, 87].
- For scientific applications with parallel loops, researchers have proposed efficient runtime scheduling algorithms using the following techniques: factoring scheduling [48], guided self-scheduling [78], and phase-based scheduling [70].

To facilitate comparison with our new approach, we present the traditional approach in terms of programming layers, as illustrated in Figure 1.1. We separate application programs and

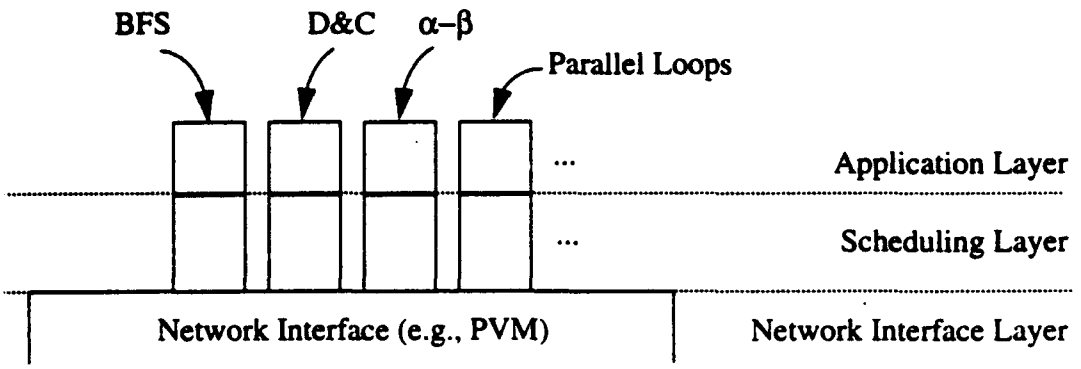


Figure 1.1: Three parallel programming layers in the traditional approach.

scheduling programs into two different programming layers, the *application layer* (high level) and the *scheduling layer* (low level), respectively. The application layer is for applications; the scheduling layer is for parallel scheduling algorithms, with each algorithm supporting one application or a class of applications. Programmers in the application layer are called *application programmers* (they are not expected to design parallel scheduling algorithms); programmers in the scheduling layer are called *scheduling programmers*. In addition to the above two layers, the *network interface layer* provides a general mechanism for network communication and supports an interface for the scheduling layer to access the network system. For example, PVM [37], Express [49], iPSC primitives[50], Nectarine [95], and socket packages for TCP/IP [65] would reside in the network interface layer. Programmers in this layer are called *system designers*. Note that in the rest of this thesis when we refer to a generic "programmer", we will mean the scheduling programmer.

The traditional approach has a serious problem: it requires a large effort to implement any of these dynamic scheduling algorithms. In order to implement a dynamic scheduling algorithm, the programmer usually needs to write the details of supportive scheduling routines, e.g. those for maintaining task lists and handling interprocessor communication for load balancing. Unfortunately, it is very difficult and time-consuming to write and to debug these supportive routines. (Note that concurrent debugging, in particular, is clearly much more difficult than sequential debugging, due to nondeterminism.) For example, when we were parallelizing a solid modeling program (called Noodles [23]) based on a simple load balancing strategy [31], it took us months to write thousands of lines of load-balancing code in C. From this, we understand that it is extremely important to provide a *general* scheduling system which can hide these supportive scheduling operations from the programmer such that programmers only

need to focus on the specification of the scheduling policies.

1.2 Our New Approach

Recently, researchers have begun to notice that these supportive scheduling operations are very similar among many scheduling algorithms. In response, some systems have begun to provide greater generality with high-level interfaces. For tree search problems, Wu [103] proposed a parallel programming system, called *dual-priority task scheduling* (an early version of our present system), which supports flexible scheduling algorithms for most tree search problems. In addition, Nishikawa and Steenkiste [70, 71] proposed the Aroma language (mainly based on phase-scheduling) which offers a range of scheduling algorithms to cover many scientific applications. Still, none of the above have been claimed to be general.

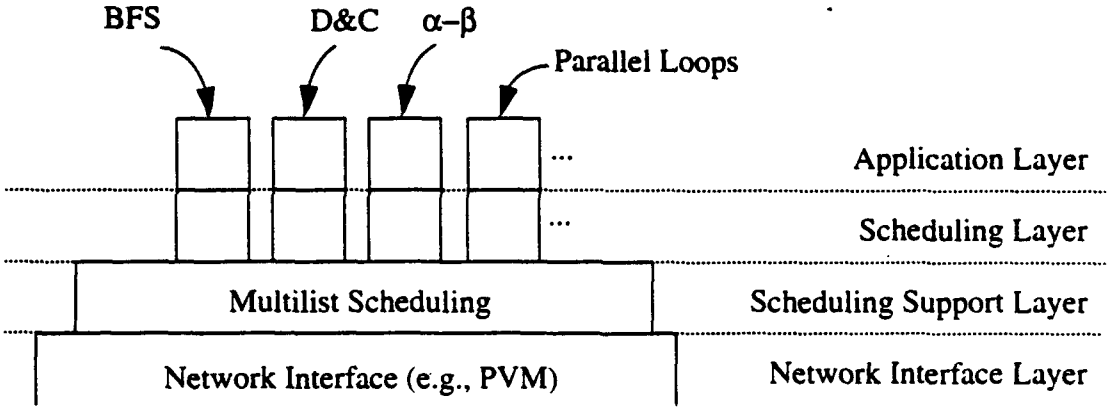


Figure 1.2: Four parallel programming layers in the traditional approach.

In this thesis, we will propose a general approach which can greatly reduce the effort of developing dynamic scheduling routines. In our new approach, we decompose task scheduling into two parts — the specification of scheduling policies and the implementation of supportive scheduling operations — and then hide the latter from the programmer. We call this approach *multilist scheduling*[2], because it is based on a uniform scheduling model involving the use of

[2]In the area of compiler design, *list scheduling* [2, 59] is a common technique in which a heuristic function is used to indicate the execution sequence of tasks (or instructions) in a scheduling list. Our system is similar to list scheduling in this sense. However, the list scheduling technique used by compilers usually sorts the scheduling list before scheduling tasks from it, while our multilist scheduling system inserts tasks into or deletes tasks from scheduling lists at runtime.

multiple scheduling lists. In the multilist scheduling model, programmers only need to specify scheduling policies based on scheduling lists. The supportive routines for the implementation of scheduling lists are moved into the system and hidden from the programmer. These supportive routines from the scheduling layer form a new layer called the *scheduling support layer* as shown in Figure 1.2. In the past, the scheduling programmers had to write the supportive routines; now, this will be the responsibility of the system designer.

1.3 Overview of This Thesis

Chapter 2 will describe our multilist scheduling model in greater detail and will also show the generality of the model. Chapter 3 will illustrate the generality and simplicity of the model by implementing several interesting scheduling algorithms, such as parallel D&C and BFS, based on the model. Chapter 4 will describe the current implementation and will show that our general approach incurs no significant performance overhead at least in these two cases. Chapter 5 will present some theoretical results which we have obtained while developing our model. Since these theoretical results have no strong relation to the rest of this thesis, the reader may skip this chapter without loss of continuity. Chapter 6 will demonstrate good performance results for some applications that are based on the parallel D&C and the parallel BFS scheduling algorithms, such as the set covering problem. Chapter 7 will give our conclusions. Appendix A will define the user interface for our multilist scheduling model.

Chapter 2

Multilist Scheduling

In Section 2.1, we will define a computational model for task scheduling. In Section 2.2, we will give a general introduction to the multilist scheduling approach. The model based on this new approach will be formally defined in Section 2.3. Finally, we will discuss this model in Section 2.4.

2.1 Computational Model for Task Scheduling

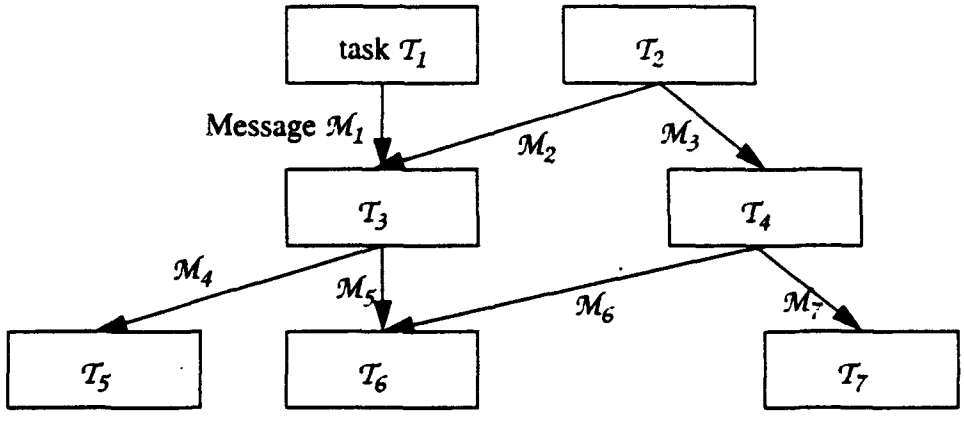


Figure 2.1: Tasks and messages.

The computational model for task scheduling has two basic elements: tasks and messages. *Tasks* are the basic program units that run concurrently; a *message* is a piece of data sent from one task to another task. Figure 2.1 illustrates an example. The computational model obeys the following rules.

1. The system creates a task *after* all the expected messages describing the task have been received. These messages will be subsequently consumed.
2. After all the expected messages have been received, a task ignores further messages.
3. A processor completes a task before switching to another task.
4. A task can generate zero or more messages. Note that for simplicity creating messages only happens at the end of the task's computation.
5. A task frees itself immediately before it is terminated.

The third rule implies that the entire computation of a task is executed sequentially on a processor, and that tasks cannot preempt each other. Therefore, if parallel computation is desired within a task, then the task should explicitly be decomposed into parallel subtasks. It also follows that this model cannot handle applications that depends on preemption, such as certain real-time applications.

Consider an example of parallel programming which requires the following primitives: (1) fork a *thread*, (2) send data to a thread, and (3) receive data in a thread. We will show that these primitives can be represented in the above computational model as follows.

Fork a thread. Create a message which subsequently creates a task corresponding to the thread. For example, in Figure 2.1, task $\mathcal{T}_3$ representing some thread can fork another thread by creating a message, say $\mathcal{M}_4$; then, message $\mathcal{M}_4$ will in turn create task $\mathcal{T}_5$, corresponding to the new thread.

Send data to a thread. Create a corresponding message (the delivery and the destination are part of the message). For example, in Figure 2.1, task $\mathcal{T}_2$ sends a message $\mathcal{M}_2$ to a thread represented by task $\mathcal{T}_3$.

Receive data in a thread. Create a message $\mathcal{M}$ (containing the content of the whole thread) for continuation of the thread; terminate the current task (corresponding to the current thread); when a send message corresponding to the receive exists, create a new task from the send message and $\mathcal{M}$ (i.e., just resume the execution of the original thread). For example, in Figure 2.1, when task $\mathcal{T}_1$ representing some thread wants to receive message $\mathcal{M}_2$, the task will create message $\mathcal{M}_1$ for continuation of the thread. Then, when message $\mathcal{M}_2$ is created, both $\mathcal{M}_1$ and $\mathcal{M}_2$ will create a new task $\mathcal{T}_3$ which resumes the execution of the original thread.

In the rest of this thesis, we will assume that a thread can receive messages at any time during its execution, but that a task can receive messages only at the beginning of its execution. So, each segment of a thread between the receipt of messages corresponds to a complete task.

In order to implement task scheduling, the programmer needs to write a procedure, called the *task scheduler*. Whenever a processor is idle, the system applies the task scheduler in order to schedule a new task, based on the state of the whole system (tasks, messages, and processors). If more than one processor tries to schedule tasks simultaneously, only one processor can apply the scheduling operation at a time. In other words, this is an *atomic* operation.

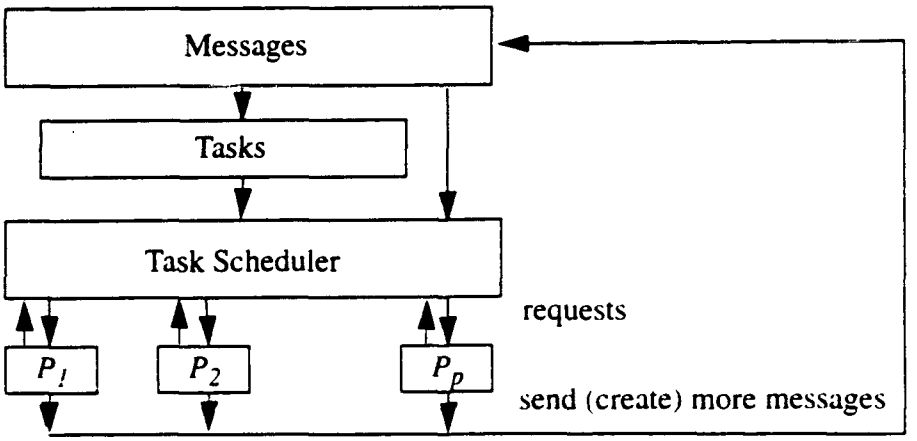


Figure 2.2: Computational model for task scheduling.

This model for task scheduling will be called the *standard scheduling model*. It is illustrated in Figure 2.2 with p processors, denoted by P_1, ..., P_p. In accordance with the definitions of scheduling programmer and application programmer in Section 1.1, the scheduling programmer is responsible for the implementation of the task scheduler, and the application programmer is

responsible for the implementation of tasks and messages. In the rest of this chapter, we will develop the model for the design of the task scheduler.

2.2 General Approach

On a single processor, a task scheduling sequence is basically a list of tasks ordered according to their *priorities*, the preferences of scheduling these tasks. We call such a list a *scheduling list*. In a parallel system, extending the above paradigm, task scheduling requires the use of multiple scheduling lists and multiple priority assignments per task. We will illustrate this necessity with the following example.

Synchronous Network Simulation

Network simulation is a common computational paradigm, in which data dependencies can be described by a network or graph. Examples of network simulation computations include finite element simulation such as fluid simulation [53]; differential equation solving such as weather prediction [24, 25]; digital circuit simulation such as gate-level simulation [20]; and digital signal processing such as sonar detection [80].

A network simulation computation can be represented by a directed graph, in which a node represents a thread and an edge represents the data dependency between threads. If there is an edge from node A to node B, node A needs to send data to node B at the end of each phase. We will call this edge an *out-edge* for A and an *in-edge* for B.

A *synchronous network simulation* (*SNS*) computation is a network simulation computation in which all nodes need to be synchronized in each phase. (Note that an *asynchronous network simulation* computation is a network simulation computation without the constraint of synchronization; we will discuss this case in Section 3.2.4.) For SNS, each node will be executed once during each phase. Then, each node will send data to neighboring nodes via the out-edges in the graph. The SNS computation can advance to the next stage when all the nodes have received data from their neighboring nodes via in-edges. (From the task definition in the previous section, each node during one phase can be considered to be a task.)

The traditional approach for SNS is to partition the SNS graph over processors by hand [62, Section 4] or by compilers [80]. However, if the grain size of each node (in a phase) cannot be known *a priori*, it becomes critical to partition the graph over processors at runtime. Here, we propose a dynamic load balancing strategy based on the criterion of keeping the number of *cross edges* as small as possible. A cross edge is an edge between two nodes, which reside on different processors.

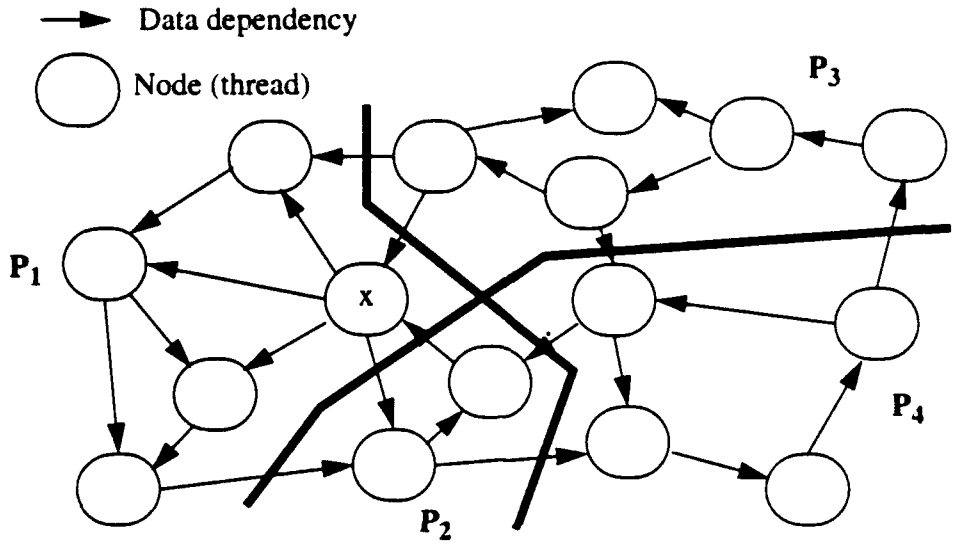


Figure 2.3: Partitioning an SNS graph over four processors.

Let us consider an SNS computation, illustrated in Figure 2.3, at some moment of a phase. Its SNS graph is partitioned over four processors, P_1, P_2, P_3, and P_4 (the partitioning is indicated by the thick lines in this figure and nodes are represented by circles). For the node marked with "x", each processor has a different preference to schedule this node: Processor P_1 has a very high preference to schedule the node because it is always good to schedule local nodes before requesting nodes from other processors. Processor P_4 has a very low preference to schedule the node because moving the task to P_4 will incur the expense of many more cross edges. Similarly, processors P_2 and P_3 have in-between preferences of scheduling the node. One can also apply the same strategy to other nodes to find the relative priority of scheduling any task on any processor. This shows that each processor can have its own perspective of the preferred task scheduling sequence. This suggests that a *general* scheduling model should at least allow each processor to express its own perspective of the task scheduling sequence.

p-List Model

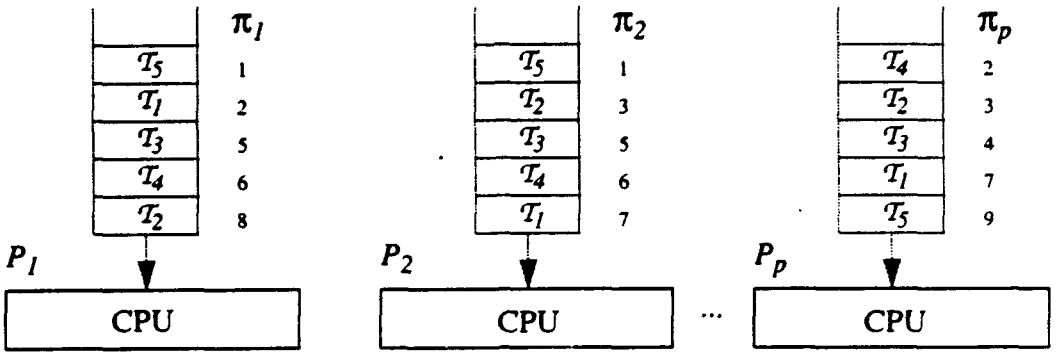


Figure 2.4: p-list model.

In a parallel system with p processors (denoted by $P_1, ..., P_p$), since each processor may have its own perspective of the task scheduling sequence, we assign, in our approach, a scheduling list to each processor. Since there are p scheduling lists in all, we call the model the *p-list model*. In this model, the programmer only needs to assign to a task p priorities, (π_1, π_2, ..., π_p), with each π_i representing the priority of the task in the scheduling list of P_i. Figure 2.4 illustrates the p-list model with five tasks denoted by $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$, $\mathcal{T}_4$, and $\mathcal{T}_5$. In this figure, the small-font number in the right hand side of each task of each list represents the priority[1] of the task in the list. For example, for task $\mathcal{T}_1$, its $\pi_1 = 2$, $\pi_2 = 7$, and $\pi_p = 7$. When a task is created, it is simultaneously inserted into each P_i's list according to the task's priority π_i. (Note that if a task is absolutely not scheduled by some processor, say P_i, then we can set the priority π_i of the task to an *undefined* value and do not insert the task into the corresponding scheduling list.) If some processor schedules a task (with the highest priority) from the head of its scheduling list for execution, say processor P_2 schedules task $\mathcal{T}_1$, all the instances of $\mathcal{T}_1$ in other scheduling lists will also be removed. If the priorities of a task are changed at runtime, the location of the task in each list must be changed accordingly.

A very important result for the p-list model is that the use of p scheduling lists is *sufficient* for specifying all parallel scheduling policies, i.e., the p-list model results in no loss of generality with respect to the standard scheduling model (as described in Section 2.1).

Assertion *The p-list model results in no loss of generality with respect to the standard scheduling model.*

[1]Note that some systems use a small number to represent a high priority but some other systems use a large number. For consistency, we will always use a large number to represent a high priority in this thesis.

Justification. This follows from the fact that we can recast any scheduling algorithm $\mathcal{A}$ in terms of our p-list model, as follows. Consider how scheduling is performed when a processor tries to schedule a task. If we use algorithm $\mathcal{A}$ (explicitly coded), the schedule is determined by executing the code; while if we use the p-list model, the schedule is determined by relative task priorities, which must be settled before task scheduling time. Since priorities may be changed dynamically in the p-list model, we only need to ensure that the highest priority in each list is always assigned to the "correct" task. To be more precise, assume that each task $\mathcal{T}_i$ is the one to be scheduled on processor P_i by algorithm $\mathcal{A}$ if processor P_i is the next processor to schedule a task. In the p-list model, the programmer can assign priorities to tasks in such a way that task $\mathcal{T}_i$ has the highest priority in P_i's list (more precisely, the ith priority of $\mathcal{T}_i$ is the highest among the ith priorities of all the tasks). Thus, whichever processor requests a task next, the p-list system will schedule the same task as algorithm $\mathcal{A}$. Thus the priority assignment scheme in the p-list model realizes algorithm $\mathcal{A}$.

The above assertion shows that the p-list model is well suited to the specification of scheduling policies. However, we do not want to naively implement each scheduling list on its corresponding processor, because such a straightforward implementation would be inefficient on a distributed-memory system, due to the following two communication problems: First, whenever a processor inserts or deletes a task, the processor would have to inform every other processor. This will result in a large amount of communication. Second, processors may compete with each other and attempt to schedule the same task simultaneously.

p^2-List Model

In order to solve these problems of excessive communication and scheduling competition (processor race conditions), we want to avoid inserting/deleting a task across processors while keeping the notion of using p scheduling lists. We will modify the p-list model as illustrated in Figure 2.5. Let the original scheduling list of each processor P_i become a *virtual list* (*VL*), denoted by VL_i, and let each P_i store p small *physical lists* (*PLs*), denoted by PL_{i1}, PL_{i2}, $\cdots$, PL_{ip}. Each individual PL_{ij} is *physically* maintained by processor P_i, while each VL_i is *conceptually* constructed from tasks appearing in PL_{1i}, PL_{2i}, $\cdots$, PL_{pi}. Since there are p^2 PLs in the entire system, we call the model the p^2-*list model*. In a VL, the tasks are merged on

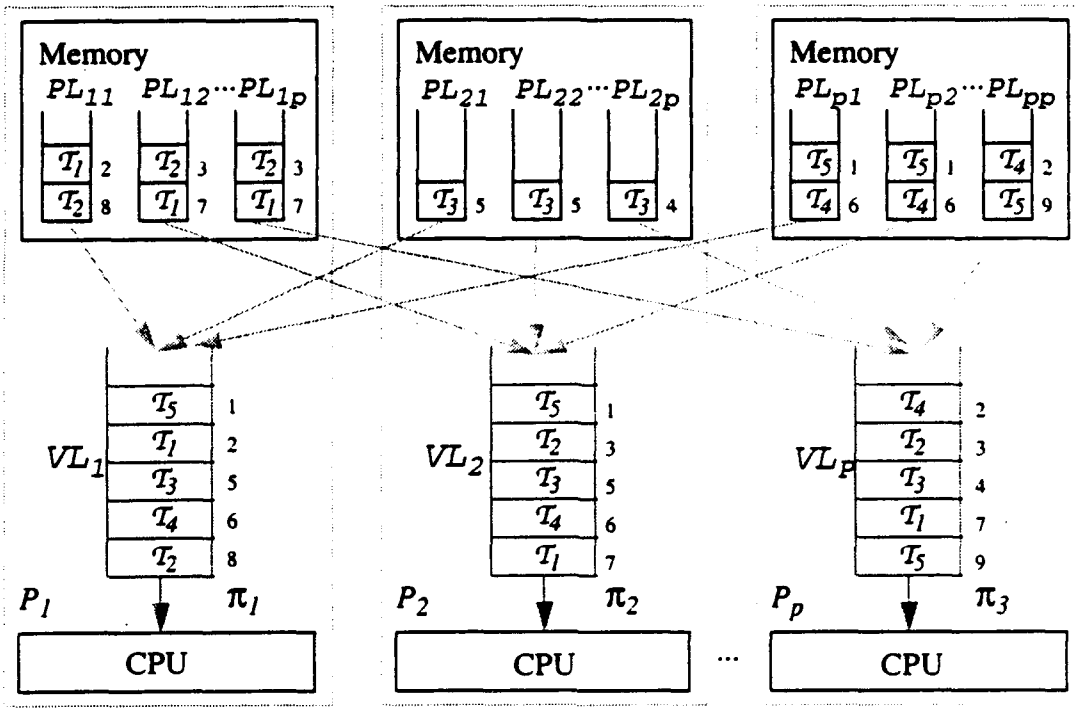


Figure 2.5: p^2-list model.

the basis of the priorities assigned to them on the corresponding PLs. When creating a task, the programmer assigns p priorities to the task and designates a processor, say P_i, where the task will be stored. (Note that in many cases the designated processor is set to the one creating the task so that the insertion operation requires no communication.) Subsequently, the system will insert the task into each PL_{ij} according to the jth priority of the task. Whenever some processor wants to schedule this task, say processor P_2 schedules $\mathcal{T}_1$ in Figure 2.5, we can delete task $\mathcal{T}_1$ from each PL_{1j}. Although the insertion and deletion operation only updates the PLs on P_i, implicitly each VL is also changed accordingly.

There are two main advantages to the p^2-list model. First, if several processors try to simultaneously schedule a task which is stored in the PLs on some processor P_i, we can resolve this by letting P_i determine which processor can schedule the task. Second, in an actual implementation of the model, each processor P_i only needs to maintain the head of VL_i, because this is the place from which processor P_i schedules tasks. (Chapter 4 will discuss this issue in greater detail.) Since tasks in the tail of VL_i are not interesting to P_i, we may eliminate the communication that would be required to maintain the tail of VL_i.

We can further improve the modified model in the following two situations: The first situation is that if some PLs on the same processor are identical, we only need to use one PL to represent these PLs. For example, in Figure 2.5, we can use one PL to stand for PL_{12} and PL_{1p}, if they are always the same.

The second situation is that if certain priorities of each task are monotonically related, then the corresponding PLs can be reduced to a single PL, supplemented with a set of monotonic functions for deriving the other PL's. For example, if the priority for each task in one PL (denoted by PL') is a monotonic function f of its priority in another PL (denoted by PL), then the system can perform operations on PL' based on the data structure of PL and the priority translation function f, as follows. (1) We only need to insert tasks into and delete tasks from PL, not PL'. (2) When some processor tries to schedule a task from PL', the system can schedule the task from the head of PL if the function f is monotonically increasing, or from the tail of PL' if f is monotonically decreasing. We call PL a *base PL* and call PL' a *derived PL*.

2.3 Multilist Scheduling Model

Now, we can modify the p^2-list model by allowing the programmer to use one list to represent another list(s), and formally define the multilist scheduling model as follows.

- On each processor P_i, the programmer creates some PLs, say k_i PLs, denoted by PL_{ij}, $1 \leq j \leq k_i$.
- For each VL_i, the programmer designates certain PLs (in the system) which are merged into VL_i. The processor P_i will automatically schedule a task from the head of VL_i, which is constructed from these designated PLs. If the scheduled task is from processor $P_{i'}$, then $P_{i'}$ will subsequently delete the task from $PL_{i'1}, \ldots, PL_{i'k_{i'}}$. Implicitly, some VLs are also changed accordingly.
- At runtime, for each task assumed to be stored on some processor P_i (which can be designated by the programmer), the programmer assigns to the task k_i priorities, denoted by $\pi_1, \pi_2, \ldots, \pi_{k_i}$. The system will insert the task into each PL_{ij} according to π_j.

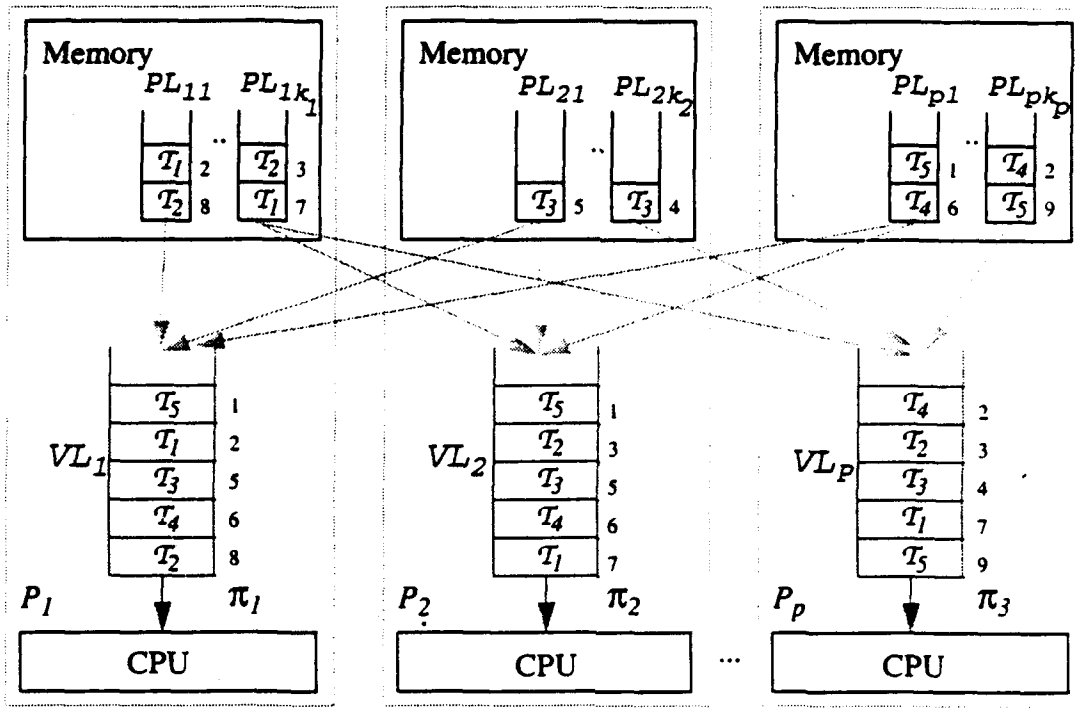


Figure 2.6: Multilist scheduling model.

Implicitly, some VLs are also changed accordingly. Note that priorities of a task can be changed dynamically.

The programmer uses the first two operations to specify a "scheduling pattern" by which the system creates PLs and merges PLs into VLs. Figure 2.6 is an example of scheduling pattern. In the p-list and p^2-list models, we have mentioned that the priorities of a task are allowed to be changed at runtime. In the multilist scheduling model, the scheduling pattern can be also changed at runtime. Although the scheduling patterns are usually fixed, we will present a good example of a scheduling algorithm (in Section 3.1.2.2) whose scheduling pattern is changed at runtime.

In addition, if a PL can be derived from another PL with a monotonic priority translation function f, the programmer only needs to specify this derivation relation for the derived PL and does not need to assign redundant priorities. The system will only insert tasks into and delete tasks from the base PL, not the derived PL. In order to schedule a task from the derived PL, the system schedules the task from the head or the tail of the base PL, depending on whether f is monotonically increasing or decreasing. Since the function f is a user-defined runtime function

and it can also depend on some runtime information, this potentially provides a simple way to dynamically change priorities, though we have not found an appropriate example to present in this thesis.

The above definition captures the essence of the multilist scheduling model. Chapter 4 will show that the addition of parameters for tasks, PLs, and VLs can further improve the performance of the system. For example, the programmer can give an estimated grain size for each task so that our system can use it to balance the load. Appendix A presents the interface for the model.

After the programmers specify the above information, they can assume that the system will schedule tasks in an order roughly corresponding to the specified priorities. However, this order may vary slightly for heuristic reasons. So, programmers should not reply on priorities for program correctness.

2.4 Discussion

If a programmer wants to implement a scheduling algorithm without the help of our model, he or she needs to do the following work.

1. Set up the scheduling pattern for the scheduling algorithm.
2. Assign priorities (and some other parameters) to each task.
3. Implement each PL in the pattern.
4. Implement the details of scheduling tasks from VLs according to the pattern.

The multilist scheduling model successfully decomposes the above work into two parts: the first two items and the last two items. Programmers only need to specify the scheduling policy in the first part, while the second part (the details of maintaining PLs and VLs) can be hidden from programmers. In fact, maintaining PLs and VLs is the most difficult part of writing a scheduling algorithm. Maintaining VLs is especially complicated due to the need for interprocessor communication.

The decomposition of the task scheduling process allows the programmers to focus on designing efficient scheduling algorithms, while allowing the system designers to focus on designing efficient supportive routines. In the former case, programmers are encouraged to design more interesting and efficient scheduling algorithms, which are hard to implement without the help of our system. Chapter 3 will illustrate several scheduling algorithms based on this model. In the latter case, the system designers can continue to develop better methods to optimize the performance of the supportive routines. Chapter 4 will propose some efficient techniques to maintain VLs and PLs for the current implementation.

In addition, we argue that it is easy for (scheduling) programmers to specify scheduling policies (items 1 and 2) based on the multilist scheduling system. When programmers come up with scheduling algorithms, the programmers usually can easily figure out the scheduling sequences of tasks (actually they are just the task sequences in VLs). Therefore, we argue that they should be able to specify the scheduling patterns and to assign priorities corresponding to the scheduling sequences. We illustrate the simplicity of implementing many scheduling algorithms based on this model in the next chapter. At least, it is less painful to implement scheduling algorithms based on our model than to write task scheduling routines from scratch.

Chapter 3

Examples of Scheduling Algorithms

This chapter will develop multilist scheduling schemes for several scheduling algorithms to demonstrate how easily the model can be utilized and how widely the model can be applied to applications. Section 3.1 will show several main examples of scheduling algorithms, each using one distinct scheduling pattern (defined in Section 2.3). Section 3.2 will show other examples of scheduling algorithms whose scheduling patterns are the same as the ones in Section 3.1. Finally, Section 3.3 will give some more discussion.

3.1 Main Examples

This section will show several main examples of scheduling algorithms, each using one distinct scheduling pattern. Section 3.1.1 will propose two multilist scheduling schemes to implement two different scheduling algorithms for parallel best-first search. In both of the schemes, each processor needs to create one physical list (PL). Section 3.1.2 will propose two multilist scheduling schemes to implement two different scheduling algorithms for parallel divide-and-conquer. In both of the schemes, each processor needs to create two PLs. Section 3.1.3 will propose a multilist scheduling scheme to implement a scheduling algorithm for parallel synchronous network simulation problems. In this scheme, each processor needs to create p PLs, if there are p processors.

3.1.1 Parallel Best-First Search

Best-first search (*BFS*) is a common computation paradigm, in which the program always schedules the best task among all possible tasks for execution. Examples of BFS computations include some *branch-and-bound* (*B&B*) problems such as the traveling salesman problem (TSP) [29], and some state-space search problems such as the 15-puzzle [75].

A BFS computation can be viewed as a process of expanding a tree. Each node in the tree corresponds to a problem instance, and children of the node correspond to its subproblems. Each node has an associated cost of its corresponding problem instance. The computation always chooses the node with the least cost for execution. A node is called a *solution node* if it represents a solution. A BFS computation tries to find among all the solution nodes the one with the least cost.

Most BFS algorithms maintain an invariant property, called *admissibility* [75], in which the cost of each node is less than or equal to the costs of its children. In these algorithms, a BFS computation terminates when it has found some solution nodes, the least cost among which is C_{min}, and it has expanded all nodes with costs smaller than C_{min}. Since the descendants of current nodes all have no smaller costs due to admissibility, we will not be able to find a solution node with a cost smaller than C_{min}. Thus, the node with cost C_{min} is the result of this computation.

3.1.1.1 Scheduling Algorithm Based on a Global Priority Queue

In order to solve BFS efficiently in parallel, we can use a simple scheduling algorithm requiring a *global priority queue* (*GPQ*), as shown in Figure 3.1(a). This scheduling algorithm will be called *PBFS-GPQ* (Parallel BFS with a GPQ) in this thesis. In this algorithm, each new node corresponding to a task is inserted into the GPQ in accordance with the cost associated with the node; each processor schedules the node with the least cost in the GPQ. Zhang [108] proved that the algorithm can balance the load very well (without considering the communication overhead).

The *PBFS-GPQ* scheduling algorithm described above can be easily implemented in the multilist scheduling model. The algorithm, whose scheduling pattern is shown in Figure 3.1(b),

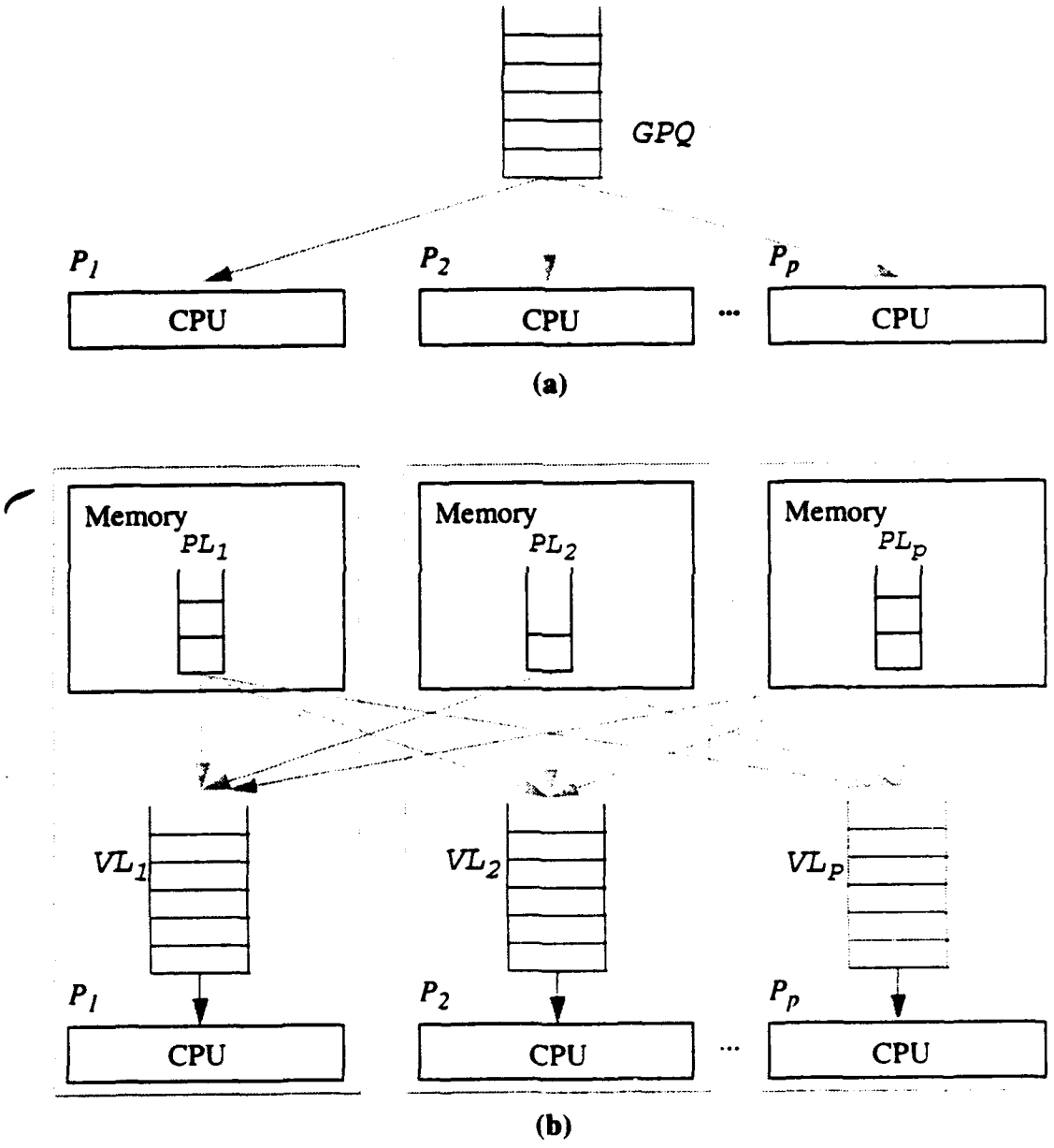


Figure 3.1: Scheduling algorithm for *PBFS-GPQ*: (a) using one GPQ and (b) using our model.

is described as follows:

- On each processor, create one PL.
- For each node (corresponding to a task) with cost C, assign a priority, $\pi = -C$. Note that minimum cost translates to highest priority.
- For each processor P_i, merge all the PLs (from all the processors) into VL_i. Thus, each virtual list (VL) is actually identical to the GPQ.

Since each processor schedules a task from its VL which is identical to the GPQ, the above multilist scheduling scheme realizes the *PBFS-GPQ* scheduling algorithm. Note that this is a basic technique to form a GPQ.

The scheduling pattern for *PBFS-GPQ* will recur throughout this chapter. We therefore find it useful to define a "global scheduling subpattern", as follows.

Definition 3.1 *A subpattern of a scheduling pattern is called a "global scheduling subpattern" if it contains at least one PL on each processor, and all its PLs are merged into all VLs.*

As for the interface to the application layer, application programmers only need to do the following two things:

- Declare initially that the *PBFS-GPQ* scheduling algorithm will be used. (This establishes the appropriate scheduling pattern in the scheduling layer.)
- Declare the cost of a node whenever one is created. (This allows the node's priority to be calculated in the scheduling layer.)

3.1.1.2 Scheduling Algorithm with Randomization

For parallel best-first search, Karp and Zhang [57] proposed a scheduling algorithm with the technique of randomization. This scheduling algorithm will be called *PBFS-R* (Parallel BFS with Randomization) in this thesis. In this algorithm, each processor has one local priority queue. Whenever a node is created, we randomly select a destination processor and then store the task into the local priority queue of that processor, according to the cost of the node. Each processor always schedules tasks for execution from its own local priority queue. Karp and Zhang also proved that this algorithm can balance the load well with a very high probability.

The *PBFS-R* scheduling algorithm described above can also be easily implemented in the multilist scheduling model, as follows:

- On each processor, create one PL.
- When a node (corresponding to a task) with cost C is created, designate a processor at random to store the task, and then assign a priority, $\pi = -C$, to the task.
- For each processor P_i, only merge its local PL_i into VL_i. Thus, each VL_i is actually identical to PL_i. So, we perform communication only when a task is created, not when a task is scheduled.

Hence, this multilist scheduling scheme above realizes the Karp and Zhang's scheduling algorithm. This is a good example of an algorithm in which a task is stored in the PLs on a different processor from the one creating the task.

3.1.2 Parallel Divide-and-Conquer

Divide-and-Conquer (*D&C*) is another common computation paradigm, in which the solution of a problem is obtained by solving its subproblems recursively. Examples of D&C computations include various sorting methods such as quicksort [43], computational geometry procedures such as convex hull calculation [79], AI search heuristics such as constraint satisfaction techniques [40], adaptive data classification procedures such as generation and maintenance of

quadtrees [86], and numerical methods such as multigrid algorithms [72] for solving partial differential equations.

A D&C computation can be viewed as a process of expanding and shrinking a tree. Each node in the tree corresponds to a problem instance; children of the node correspond to its subproblems. During the computation, each internal (non-leaf) node goes through two phases. The first phase is the *divide phase* during which the problem instance associated with the node is divided into subproblems. The second phase is the *combine phase* during which the solution of the problem instance associated with the node is derived by combining solutions of the subproblems associated with the node's children. Each leaf after its creation will perform some computation and return the results to its parent. At a given time, nodes on a wavefront that cuts across all paths from the root to leaves can be active in performing divide, combine, or compute operations. Along each path the wavefront first moves down from the root to its leaf and then up from the leaf to the root. For simplicity of discussion, we will ignore the shrinking phase. It will be useful to define a *frontier node* as a node which has been generated but has not been expanded and a *local frontier node* of a processor as a frontier node whose parent was expanded (or executed) on this processor.

3.1.2.1 Wu-Kung's Scheduling Algorithm

In order to perform D&C efficiently in parallel, we [105] designed an efficient scheduling algorithm, called *PDC-WK* (Parallel D&C with Wu and Kung's method) in this thesis. The *PDC-WK* scheduling algorithm schedules nodes according to the following rules. First, if a processor has local frontier nodes, it must schedule the deepest among them. This is the *depth-first search* which can minimize the local memory requirement and avoid wasteful interprocessor communications. Second, when a processor runs out of local frontier nodes, it follows *breadth-first* search to schedule a frontier node closest to the root, from all (other) processors. Note that the node closest to the root is likely to contain the largest subtrees, which will have the most locality and therefore will need the least communication.

We also proved that, among all the scheduling algorithms which can split the load nearly evenly, our algorithm is optimal with respect to the *communication cost*, which is defined to be equal to the total number of *cross nodes*. (A cross node is a node which is generated by one processor but expanded by another processor.) A more detailed description for the above

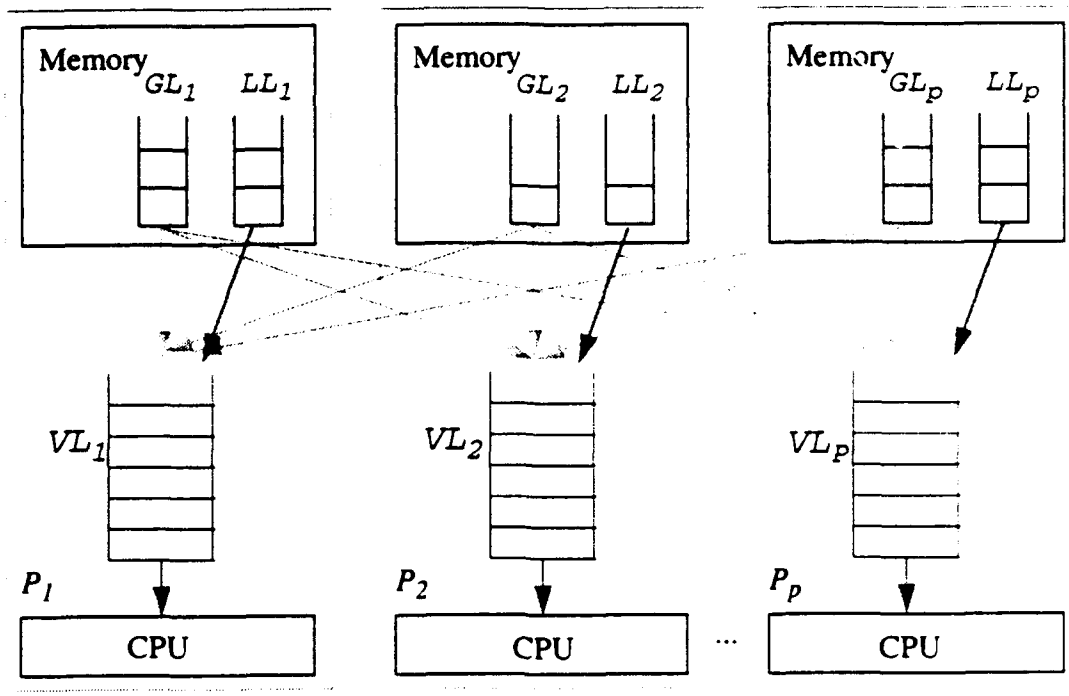


Figure 3.2: Scheduling pattern for *PDC-WK*.

results will be presented in Section 5.1.

The *PDC-WK* scheduling algorithm described above can be easily implemented in the multilist scheduling model. The algorithm, whose scheduling pattern is shown in Figure 3.2, is described as follows.

- Create two PLs, *local list* (*LL*) and *global list* (*GL*), on each processor.
- For each processor P_i, merge all the GLs (from all the processors) and its own LL into VL_i. According to Definition 3.1, this scheduling also includes a global scheduling subpattern like *PBFS-GPQ*, but each processor may also schedule nodes from its own LL.
- For each node (corresponding to a task), assign to it two priorities: local priority $\pi_L = l$ (corresponding to LL) and global priority $\pi_G = -l$ (corresponding to GL), where l is the level of the node in the tree. A node is said to be at tree level i if it is the ith node on the path from the root to the node. The root is at tree level 0.

The above multilist scheduling scheme satisfies the scheduling rules of *PDC-WK*, as we shall now show. Since $\pi_L = l$, the processor always follows depth-first search to schedule a local node, if one exists; otherwise, because $\pi_G = -l$, the processor always schedules the node closest the root from processors. Thus, this scheme realizes the *PDC-WK* scheduling algorithm.

Since $\pi_L = -\pi_G$ in the above scheduling scheme, we can further improve it by letting LL be a derived PL based on GL with the priority translation function $f(\pi) = -\pi$. This is a good example showing that derived PLs can be used to optimize the performance.

As for the interface to the application layer, application programmers need to do the following two things.

- Declare initially that the *PDC-WK* scheduling algorithm will be used. (This establishes the appropriate scheduling pattern in the scheduling layer.)
- Declare the tree level l of a node whenever a new node is created. (This allows the node's priorities π_L and π_G to be calculated from l in the scheduling layer.)

3.1.2.2 Scheduling Algorithm Based on the Round-Robin Strategy

For parallel D&C, some researchers in [34, 44, 82] have used another scheduling algorithm based on a round-robin strategy. This scheduling algorithm, called *PDC-RR* (Parallel D&C with the Round-Robin strategy) in this thesis, is the same as the *PDC-WK* scheduling algorithm except for the following: in *PDC-RR* an idle processor will try to schedule the node closest to the root among those on its *pre-selected* processor (not in the whole system as *PDC-WK*), which is dynamically changed in a round-robin fashion as follows. Each processor has a variable s, representing the ID of the pre-selected processor. When the processor is idle, it requests a frontier node from processor P_s (as above) and lets the next s be $(s \bmod p) + 1$ such that the next node request will be sent to the next processor $P_{(s \bmod p)+1}$. One possible drawback for *PDC-RR* is: since the node scheduled by the idle processor is closest to the root only on P_s (not globally), some other processors may have nodes much closer to the root, i.e., it is very likely that the scheduled node has not the largest subtree.

Since the difference between the *PDC-RR* and *PDC-WK* scheduling algorithms is in the set of processors from which an idle processor will schedule a node, we can implement *PDC-RR* by modifying *PDC-WK* as follows: For each processor P_i, merge the GL on processor P_{s_i} and the LL on P_i into VL_i, where s_i is a variable on P_i. After processor P_{s_i} is requested, the variable s_i is changed to $(s_i \bmod p) + 1$. Thus, this above scheduling policy based on the multilist scheduling model realizes the *PDC-RR* scheduling algorithm. This scheduling algorithm is a good example showing that the scheduling pattern may be changed dynamically.

3.1.3 Parallel Synchronous Network Simulation

Section 2.2 defined the *Synchronous Network Simulation* (*SNS*) problem by using a graph to represent it. In that section, we also mentioned a dynamic scheduling strategy based on the criterion of keeping the total number of cross edges (also see the definition in that section) as small as possible. In this section, we will first describe the scheduling algorithm based on the dynamic scheduling strategy and then propose a corresponding multilist scheduling scheme.

In this scheduling algorithm, we partition the graph of nodes (each corresponding to a thread) over the processors at the beginning of the computation. Then, during each phase, we dynamically balance the load as follows:

1. If a processor has some local nodes (residing on the processor) that have not been executed yet, it always schedules the local nodes which are connected to no cross edges; then, it schedules the local nodes which are connected to some cross edges.

2. When a processor has executed all local nodes, it schedule a node from another processor such that the total number of cross edges increases the least.

The scheduling algorithm described above can be easily implemented in the multilist model. The algorithm, whose scheduling pattern is the same as the one in Figure 2.5, for a p-processor system is described as follows:

- Create p PLs on each processor. Let PL_{ij} denote the jth PL on processor P_i.

- For each processor P_i, merge all the ith PLs (i.e., PL_{ji} for all j) into VL_i.
- Assign p priorities to a node on P_i as follows:
  - The jth priority of a node ν, $j \neq i$, is $\pi_j = E_{\nu,j} - E_{\nu,i}$, where $E_{\nu,k}$ is the number of edges (including in-edges and out-edges) between ν and all the nodes on P_k.
  - If a node ν is connected to some cross edges, its ith priority π_i is set to $E_{max} + 1$, where E_{max} is the maximum number of edges which each node can have; otherwise, the priority is set to an even higher priority, say $E_{max} + 2$.

One can verify that the above multilist scheduling scheme satisfies the two scheduling rules for parallel SNS.

We note that this scheduling algorithm needs to change priorities at run time. When a node is moved to another processor, some $E_{\nu,i}$ values may be changed and therefore the priorities of the node's neighbors will be changed accordingly. This is a good example showing the need of changing priorities dynamically.

As for the interface to the application layer, *application programmers only need to do the* following:

- Initialize the scheduling algorithm for parallel SNS by describing its graph of nodes, corresponding to threads or tasks. (This establishes the appropriate scheduling pattern, and allows the priorities of the nodes to be calculated from their environment in the graph, in the scheduling layer.)

3.2 Other Examples of Scheduling Algorithms

This section will present some more examples of scheduling algorithms: the factoring algorithm for parallel loops in Section 3.2.1, the principle variation splitting algorithm (slightly modified from [21, 69]) for α-β search in Section 3.2.2, the scheduling algorithm for parallel quicksort in Section 3.2.3, and the scheduling algorithm for parallel asynchronous network simulation in Section 3.2.4. The scheduling patterns for these scheduling algorithms are the same as the ones used in the previous section.

3.2.1 Parallel Loops with the Factoring Technique

Parallel loops (without dependencies between their iterations) are very rich resources where we can exploit parallelism in many applications, especially for scientific applications. Since the amount of computation in each iteration may not be fixed, an efficient algorithm needs to balance the load at runtime while minimizing the amount of communication.

Hummel *et al.* [48] proposed an efficient runtime technique, called *factoring*. Consider how to parallelize a parallel loop with m iterations on p processors. Without loss of generality, we assume m to be $p(2^k - 1)$, where k is an integer. In the factoring technique, iterations are first grouped into tasks such that there are p tasks each of which will execute 2^{k-1} iterations, p tasks each of which will execute 2^{k-2}, ..., and p tasks each of which will execute one iteration. Then, each processor always schedules the task with the largest number of iterations next because the fine grained tasks should be preserved for better load balancing near the end. Since the total number of tasks is pk, we schedule tasks at most pk times, which is a very small number when compared with m.

We can implement this algorithm in our multilist model by simply using the scheduling pattern of the *PBFS-GPQ* scheduling algorithm and letting each task with 2^i iterations have the priority i.

Nishikawa [70] has recently suggested the following modification to further reduce communication in a distributed-memory system. Initially, we evenly distribute these tasks over the processors such that each processor has *one* task with 2^{k-1} iterations, *one* task with 2^{k-2} iterations, ..., and one task with one iteration. Basically, each processor executes its own tasks based on the same strategy: schedule the task with the largest number of iterations first. However, when a processor (say P_1) "falls far behind" another processor (say P_2), P_1 will "off-load" some task to P_2. To be more precise, let us consider the situation in which processor P_1 is the *slowest* processor which has the task with 2^i iterations and processor P_2 is the *fastest* processor which has the task with 2^j iterations, where $i > j$. The programmer can decide a threshold t (a positive integer). Then, if and only if $i - j > t$, processor P_2 will schedule the task with 2^i iterations from P_1.

Nishikawa's scheduling algorithm can also be easily implemented in our multilist model by using the scheduling pattern of *PDC-WK*. Initially, we partition tasks as above and then assign

priorities to each task with 2^i iterations as follows: the local priority $\pi_L = i$ and the global priority $\pi_G = i - t$. For the situation given in the previous paragraph, the task with 2^i iterations on processor P_1 has a global priority $i - t$ while the task with 2^j iterations on P_2 has a local priority j. If $i - j > t$ (i.e., $i - t > j$), then P_2 will schedule the task with 2^i iterations from P_1. Otherwise, no load balancing is required. Thus, this multilist scheduling scheme is the same as Nishikawa's scheduling algorithm.

Since $\pi_L = \pi_G + t$, we can improve the scheduling scheme by letting LL be a derived PL based on GL with the priority translation function $f(\pi) = \pi + t$. This is another example (*PDC-WK* was the first example) showing that we can use derived PLs to optimize the performance. The priority translation function here is monotonically increasing while that for the *PDC-WK* scheduling algorithm is monotonically decreasing.

3.2.2 Parallel Alpha-Beta Search with Principle Variation Splitting Algorithm

Alpha-beta (α-β) search [58] is a common computational paradigm for two-player game search problems, e.g., Chess [91] and Othello [64]. An α-β computation can also be viewed as a process of expanding/shrinking a tree, as in D&C in Section 3.1.2, but with the following properties.

- Each leaf node has an estimated heuristic value and returns this value to its parent.
- Each internal node in the divide phase expands some children and sorts them according to how likely they are to contain an optimum heuristic value in the subtree rooted at the child. We will let the *leftmost child* be the most promising child.
- Each internal node in the combine phase receives all the values returned from its children and then returns the maximum of the negatives of these values.
- The solution of a game tree is the returned value of the root and the identity of the child who has the negative of that value.

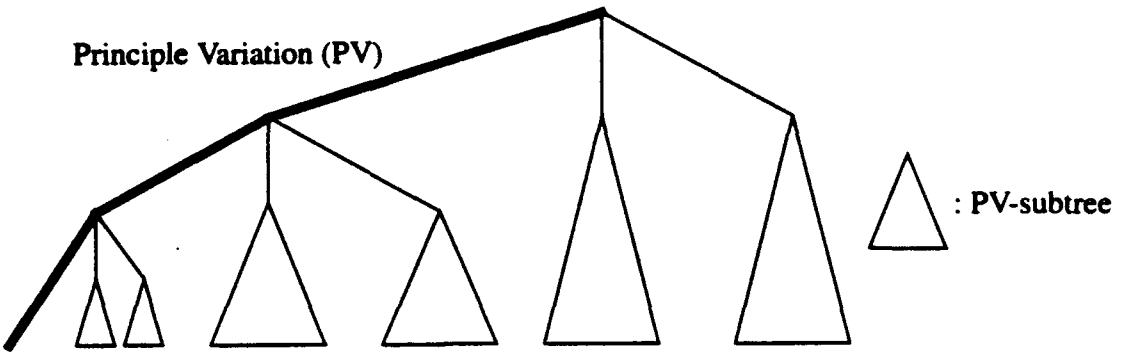


Figure 3.3: PV-subtrees.

In order to solve α-β efficiently in parallel, some researchers [21, 69] have proposed the *principle variation splitting* (*PVS*) algorithm. In the PVS algorithm, we search the path from the root to the leftmost leaf, in the first stage. The path is called *principle variation* (*PV*) and nodes on PV are called *PV nodes*; in addition, a subtree is called a *PV-subtree*, as shown in Figure 3.3, if its root is a child of a PV node but the root itself is not a PV-node. Then, in the next stage, PV-subtrees rooted at the deepest tree level are split among processors. After this stage completes, subtrees rooted at the second deepest tree level are split among processors in the next stage, and so on. These researchers used the so-called tree splitting algorithm in [69] to split subtrees among processors. But, here, we will simply use the *PDC-WK* scheduling algorithm (described in Section 3.1.2) to split subtrees among processors so that we balance the load of nodes in these subtrees while minimizing the communication. Although the original PVS algorithm goes through the stages serially, our scheduling algorithm for PVS does not have to. That is, if there are no available nodes for the current stage, idle processors can schedule the nodes for the next stage.

Now, we want to design the PVS algorithm based on the multilist scheduling model as follows. The multilist scheduling scheme for PVS uses the same scheduling pattern as *PDC-WK*. Note that each processor has two physical lists, the local list (LL) and the global list (GL). (For simplicity of discussion, we omit the combine operation in α-β search, as we did for D&C.) We assign priorities to each node according to the following rules.

- For each node on PV, its local priority is $\pi_L \doteq \pi_{max}$ and its global priority is $\pi_G = \pi_{max}$, where π_{max} is the priority larger than any of the priorities assigned below.

- For each node (corresponding to a task) at tree level i and in a PV-subtree rooted at tree level j, let the local priority be $c * j + i$ and the global priority be $c * j - i$, where $c > 2h$ and h is the tree height.

From the first rule, executing the nodes in PV is the first priority. From the second rule, since $c > 2h$ and $h \geq i$, j represents the primary key and i represents the secondary key. Since j is the primary key, we search PV-subtrees rooted at a deeper tree level earlier. For the PV-subtrees rooted at the same tree level, the value j is the same and therefore the scheduling algorithm is just the same as *PDC-WK*. Thus, the above scheduling scheme realizes our PVS algorithm.

As for the interface to the application layer, application programmers only need to do the following things.

- Declare initially that the PVS scheduling algorithm will be used (this will establish the scheduling pattern of *PDC-WK* in the scheduling layer); then, give the maximum tree height h.
- For each node (corresponding to a task), declare the tree level i of the node and the tree level j of the PV-subtree that includes the node. (This allows the node's priorities π_L and π_G to be calculated from i, j and h in the scheduling layer.)

3.2.3 Parallel Quicksort Algorithm

Sorting is the most common operation for data processing. Given an array of n elements each containing a key (not ordered), the problem is to sort the elements in the array according to the key values.

Quicksort [43] is a fast sorting algorithm, based on the divide-and-conquer technique, and with an average computation time of[1] $O(n \log n)$. This algorithm is described as follows:

- Pick the first element (or pick one at-random). Let k be the key value of this element.

[1] If $g(n) = O(f(n))$, there exists some positive c for which $g(n) \leq cf(n)$ for all sufficiently large n.

- Partition the array into two subarrays of elements such that the key values of elements in one subarray are less than k and the key values in the other subarray are greater than or equal to k.

- Recursively sort each subarray.

Since the quicksort algorithm is based on the D&C technique, we can, of course, apply the *PDC-WK* scheduling algorithm to the quicksort algorithm. However, based on some characteristics of quicksort, we suggest different priority assignment for possibly improving the *PDC-WK* scheduling algorithm.

In *PDC-WK*, since it is assumed that the shape of a tree (or subtree) cannot be predicted *a priori*, we can only use the tree depth of a node to roughly estimate the average computation amount (or locality) of a node and then use it to evaluate the global priority. However, for the quicksort problem, the average time complexity of a node, corresponding to a sorting subproblem with an array of n' elements, is $O(n' \log n')$. Since a node with a larger value of n' will typically require more computation, an idle processor wants to schedule the node with the largest n' among all the other processors. So, we can use $n' - n$ to represent the global priority of the node, instead. Note that the item $-n$ is added to the global priority to ensure that the global priority is non-positive and, therefore, less than or equal to the local priority. As for the local priority, since each processor may want to sort a small local array first in order to preserve the large array tasks for later donation (note that donating a task with greater locality may reduce the communication amount as mentioned earlier), we can let the local priority be $n - n'$.

One potential problem for the above algorithm is that the value of n' can be any number between 0 and n and therefore there could be too many distinct priorities, which will result in more task scheduling overhead (this will be discussed in greater detail in the next chapter). To avoid this problem, we can squash the range of priorities, by letting the local priority of a node be[2] $\lceil \lg(n/n') \rceil$ and the global priority of the node be $\lceil \lg(n'/n) \rceil$. This reduces the number of distinct priorities to at most $2 \lg n$, while roughly preserving their relative sequence.

[2] $\lg x = \log_2 x$. And, $\lceil x \rceil$ is the smallest integer larger than or equal to x.

3.2.4 Parallel Asynchronous Network Simulation

In Section 3.1.3, we discussed the synchronous network simulation problem. However, some network simulation problems do not have to be synchronized at the end of each phase. In those problems, as long as a node (thread) has received all of its coming data via its in-edges, the node can advance to the next phase and continue to execute. We call such problems *asynchronous network simulation* (*ANS*).

For parallel ANS, we can modify the scheduling algorithm for parallel SNS (in Section 3.1.3) by redefining the ith priority π_i' of a task (corresponding to a process in some phase) as follow: let the primary key be the negative of the phase index[3] and the secondary key be the original π_i. In accordance with the primary key, the new algorithm will try to schedule all the tasks in the current phase before advancing to the next phase. Thus, the scheduling algorithm basically is the same as the scheduling algorithm for parallel SNS except that some processors may be able to start executing some tasks for the next phase while waiting for tasks (from other processors) for the current phase.

The above scheduling algorithm still has one problem: we often need to move tasks over processors to balance the load near the end of each phase. We can reduce the communication for load balancing by balancing the load only when one processor falls far behind another processor. For example, suppose that the fastest processor, say P_i, has some tasks in a phase ϕ_i, while the slowest processor, say P_j, has no tasks in a newer phase than ϕ_j, where $\phi_i > \phi_j$. If $\phi_i - \phi_{thr} > \phi_j$, we say P_j falls far behind P_i, and then P_j can schedule some task from processor P_i, where ϕ_{thr} is a positive threshold given by the programmer.

To implement the above modified scheduling algorithm, we only need to change some priorities as follows. For each task on each processor P_i, we add ϕ_{thr} to the primary key of its ith priority. Since the ith priority of the task stands for the priority of scheduling the task locally, a processor tends to schedule local tasks with high priorities unless another processor falls far behind.

[3]The index of a phase is the index of its previous phase plus one.

3.3 Discussion

The multilist scheduling schemes proposed in this chapter demonstrate that our model can be widely applied to many scheduling algorithms. These examples also demonstrate that it is much easier to implement scheduling algorithms using multilist scheduling than to write sophisticated scheduling routines from scratch. For example, in our experiments, the code for the *PDC-WK* and *PBFS-GPQ* scheduling algorithms (shown in Appendix A.2) only has about 10-20 lines. A program of this size can be written within tens of minutes. This is in sharp contrast with previous dynamic load balancing programs, which would typically require thousands of lines of C code. This was the case in our earlier experience [31, 62] in parallelizing Noodles, a solid modeling program [23]. It took us months to write the load balancing part! Since our approach can greatly shorten the time of implementing a scheduling algorithm, we expect more interesting and complicated scheduling algorithms to be devised and implemented.

Although this chapter shows the simplicity of implementing a multilist scheduling scheme for a given scheduling algorithm, we offer no advice on how to come up with the scheduling *algorithm itself*. This is because there are too many factors which can affect system performance. These factors includes, for example, the relative importance of increasing parallelism, minimizing the total amount of computation, minimizing the required amount of communication, reducing the memory requirement, and reducing the number of distinct priorities. We find it difficult to provide a simple rule for assigning priorities based on all the factors. So, we leave this problem open. We only want to argue that it is simple to implement a multilist scheduling scheme for a given scheduling algorithm, and (in the next chapter) that our general approach incurs no significant performance overhead. In addition, since it is so easy to vary a scheduling algorithm in our model by simply adjusting priorities or scheduling patterns, it becomes easy for a programmer to find the best scheduling algorithm by searching the design space empirically.

Chapter 4

Implementation Issues

In addition to the issues of simplicity and generality of the multilist scheduling model (as discussed in the previous chapters), another important issue is to *efficiently* implement this model such that our general approach incurs no significant performance overhead. Sections 4.1 and 4.2 will respectively propose some efficient techniques of implementing virtual lists (VLs) and physical lists (PLs), on which the model is based. Section 4.3 will show that our general approach incurs no significant performance overhead at least for certain important scheduling algorithms.

4.1 Maintaining Virtual Lists

Since VLs are conceptually constructed from PLs which may be on different processors, we need to maintain VLs via interprocessor communication. Section 4.1.1 will first describe the *standard protocol*, which we can use to merge PLs into VLs in *all* cases. Section 4.1.2 will describe a more efficient protocol, called the *global protocol*, which we can use when the scheduling pattern includes a global scheduling subpattern (as defined in Definition 3.1).

4.1.1 Standard Protocol

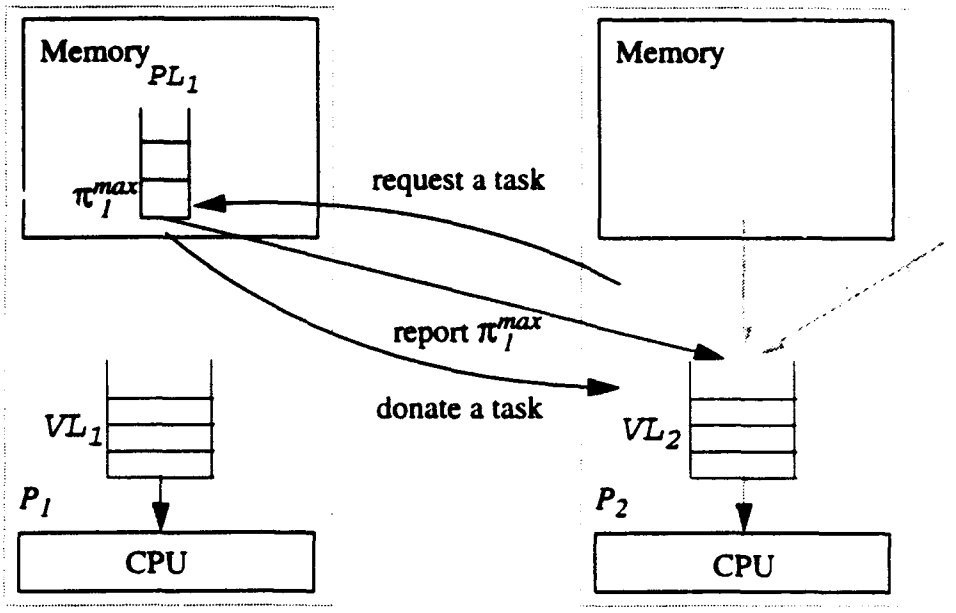


Figure 4.1: An example of the standard protocol (PL_1 is merged to VL_2).

The standard protocol is applicable whenever there is a need to merge a PL into a VL over a network. Figure 4.1 illustrates an example in which a PL, say PL_1 on processor P_1, is one of the PLs merged into a VL, say VL_2 on another processor P_2. Let π_1^{max} denote the highest priority in PL_1. We can straightforwardly implement the standard protocol as follows.

- Whenever π_1^{max} is changed, P_1 reports the new π_1^{max} to P_2.
- P_2 requests a task from PL_1 if π_1^{max} is higher than the priorities in any other PLs merged into VL_2. Then, P_1 donates a task from PL_1 to P_2 (and removes the task from all other PLs on P_1).

In the standard protocol, it may turn out that a PL which is merged into a VL may need to report to the processor with the VL too frequently. In order to reduce the number of reports to achieve better performance, we allow the programmer to provide some more information in the following two ways.

- The programmer can describe the known range of priorities in each PL. Consider the scheduling pattern in Figure 4.2, in which the priority range of PL_1 is between π_l and

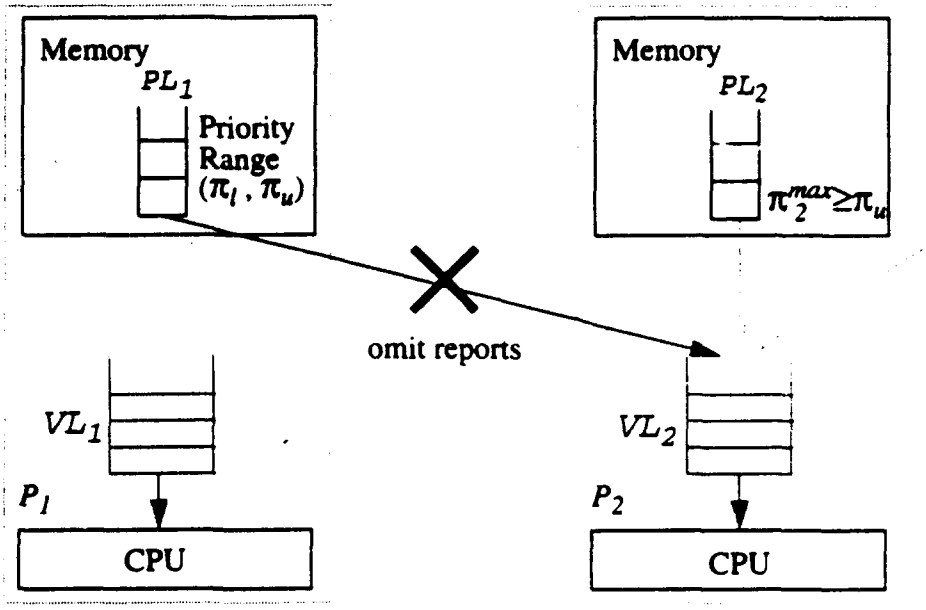


Figure 4.2: Omitting reports based on given priority ranges.

$\pi_u (\geq \pi_l)$. When the priority of some task in PL_2 is greater than or equal to π_u, VL_2 can disable all reports from PL_1 because processor P_2 will not need to schedule a task from PL_1. For example, in the *PDC-RR* scheduling algorithm, since each local list LL contains only non-negative priorities, and each global list GL contains only non-positive priorities, a processor does not need to report the maximum priority of its GL to any other processors. A processor will request a task from a GL only when the processor becomes idle.

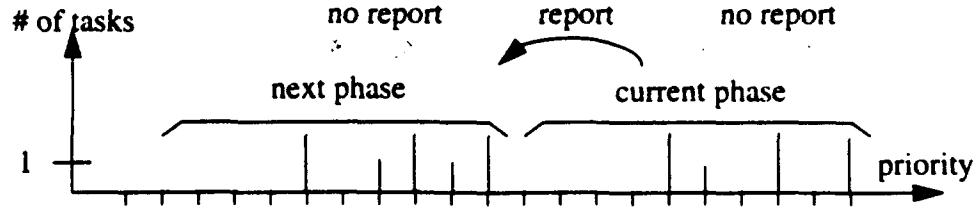


Figure 4.3: Indivisible ranges for parallel ANS (no report when the phase index is still the same).

- The programmer can define indivisible priority ranges for each PL, such that all the priorities within one range are considered as one priority value by other processors.

Thus, the PL need not report when updating π_i^{max} within the same range. For example, for the parallel asynchronous network simulation (ANS) described in Section 3.2.4, we need to balance the load mainly when the primary priority, the phase index, is changed. Since it is not important to let the other processor know the secondary priority, the programmer can declare all priorities with the same phase index to lie within the same priority range, so that PLs will avoid making reports until the phase changes, as shown in Figure 4.3.

4.1.2 Global Protocol

In some situations, we can achieve better performance by using a different protocol. In particular, let us consider the case where the scheduling pattern includes a global scheduling subpattern (defined in Definition 3.1). The case is important for many scheduling algorithms (in Chapter 3), e.g., *PBFS-GPQ*, *PDC-WK*, the quicksort algorithm, the factoring scheduling algorithm, and the principle variation splitting algorithm. For simplicity of discussion, we consider that in the global scheduling subpattern each processor P_i has one and only one PL, denoted by PL_i, which is merged into each VL (see Figure 3.1(b)). Let π_i^{max} denote the highest priority in PL_i and π^{max} denote $max(\pi_i^{max})$ for all i. For a scheduling pattern that includes the global scheduling subpattern, the standard protocol may be inefficient in the following situations:

- Whenever the value of π_i^{max} is changed, processor P_i will broadcast a message to all VLs even if some other π_j^{max} is already higher than π_i^{max}. The broadcast may result in unnecessary communication overhead.
- Many processors may simultaneously send task requests to the processor containing the task with π^{max}. This processor which receives these requests will "off-load" more than one task. Among these offloaded tasks, perhaps, only the first task has a high priority. For example, PL_1 has tasks with priorities (4, 6, 8, 9) and PL_2 has tasks with priorities (1, 2, 10). If P_3 and P_4 need to request tasks at the same time, they both will send task requests to P_2. Thus, either P_3 or P_4 will get the task with a low priority of 2.

In order to improve the efficiency when there is a global scheduling subpattern, we will introduce a central mechanism to help regulate the scheduling protocol. This mechanism is

called a *global load balancer* (*GLB*), and this kind of scheduling protocol is called a *global protocol.*

In Section 4.1.2.1, we will first propose a simple global protocol to perform load balancing by considering only those tasks with the highest priority π^{max}. If there are enough tasks with priority π^{max}, we can ignore the overhead that task prioritization causes. However, if there are not enough such tasks, task prioritization may result in significant overhead. We will describe a solution to this problem in Section 4.1.2.2 and describe an advanced protocol based on the solution in Section 4.1.2.3.

4.1.2.1 Simple Global Protocol

For a global protocol, the GLB basically wants to perform load balancing by considering only those tasks with the highest priority π^{max}. Let $\top_0$ be the set of tasks with π^{max}. (Later, we will let $\top_1$ be the set of tasks with $\pi^{max} - 1$.) The GLB needs to keep track of π^{max} and $\top_0$ in order to balance the load. So, if a processor sends a task request to the GLB, the GLB can decide which task in $\top_0$ to donate.

In the global protocol, the GLB needs to monitor each processor's status and keep track of π^{max}. Each processor P_i has the following status:

- π_i^{max}. The highest priority in PL_i.
- π^{max}. The highest priority in the entire system. It must be obtained from the GLB because only GLB can gather all π_i^{max} together to determine the value. Since some other PLs (e.g., the local list for *PDC-WK*) may also be merged into VL_i, P_i needs to compare π^{max} with priorities in those PLs each time when scheduling tasks from its VL. Therefore, it would be more efficient for each processor to have the value π^{max} locally.
- $L_i^{\top_0}$: The load of those tasks in $\top_0$ and on P_i. The *load* of a given set of tasks $\top$ is $\sum_{\mathcal{T} \in \top} G_{\mathcal{T}}$, where $G_{\mathcal{T}}$ is the grain size of $\mathcal{T}$. Basically, the grain size of a task represents the amount of computation for the task, given by the programmer. The grain size will be defined more precisely in Section 4.1.2.3. The GLB can balance the load by donating tasks from $\top_0$ to an idle processor.

- $L_i^{\mathsf{T}_1}$: The load of those tasks in T_1 and on P_i, where T_1 is the set of tasks with priority $\pi^{max} - 1$. The GLB can balance the load by donating tasks in T_1 while T_0 is empty. The purpose of having T_1 is to make the transition of load balancing from T_0 to T_1 smoother. If we do not do this, the GLB cannot respond to the idle processor while T_0 is empty, and therefore the idle processor will keep idling.

Now, we can describe a simple global protocol as follows.

1. The system balances the load round-by-round, where a "round" is defined as a period of time in which π^{max} remains the same.
2. Each processor P_i reports π_i^{max} and the changes of $L_i^{\mathsf{T}_0}$ and $L_i^{\mathsf{T}_1}$ to the GLB, in either of the following two situations.
   - $L_i^{\mathsf{T}_0}$ or $L_i^{\mathsf{T}_1}$ is changed "significantly" (e.g., by a factor of two), or
   - π^{max} is changed (i.e., a new round is started).
3. When π^{max} is changed, the GLB broadcasts the new value of π^{max} to each processor to start a new round. Note that π^{max} decreases only when no more tasks have priorities $\pi \geq \pi^{max}$.
4. When a processor requests a task, the GLB tries to balance the load by donating a task in T_0 to the processor. If T_0 is empty, the GLB will try to donate a task in T_1, instead.

An important feature of the above protocol is that an idle processor can request a task via as few as three "hops": (1) the idle processor issues a task request to the GLB, (2) the GLB forwards the message to the donor (selected by the GLB), and (3) the donor donates a task to the idle processor.

Although the GLB is conceptually centralized, it actually uses a tree structure, which can be distributed over processors as illustrated in Figure 4.4 (note that this need not be a binary tree). The main purpose is to prevent a single GLB process from becoming a bottleneck, especially when the number of processors is large (say 1000). For example, in the global protocol, broadcasting and collecting information would make a single GLB become a bottleneck given a large number of processors. An extra advantage for such a tree structure is that load balancing

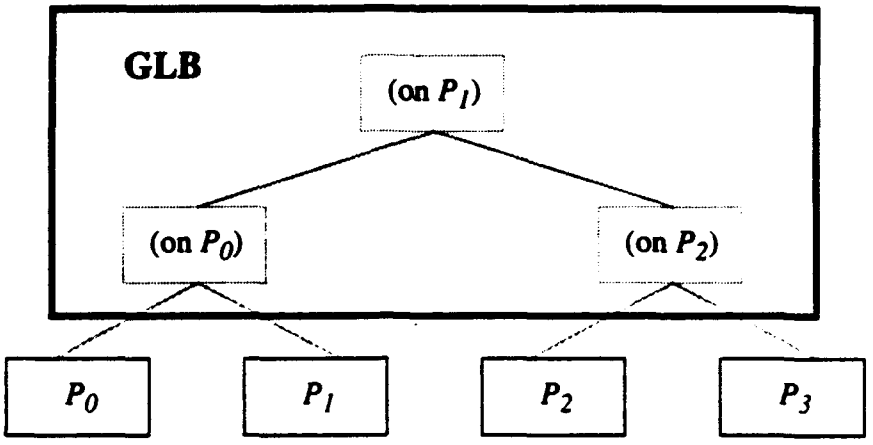


Figure 4.4: An example for the GLB hierarchy.

may happen in parallel. For example, as shown in Figure 4.4, if processors P_0 and P_2 have some tasks in T_0 and processors P_1 and P_3 simultaneously request tasks, P_0 and P_2 can respectively donate tasks in T_0 to P_1 and P_3 at the same time. In this case, each task request requires three "hops".

Definition 4.1 *In a processor tree (as illustrated in Figure 4.4), a* COMBINING OPERATION *is defined as follows. Starting from the leaf processor nodes, each node other than the root sends one packet upwards to its parent. If the processor node is an internal processor node (other than the root), it must receive all the packets from its children before sending its packet to its parent.*

A DISSEMINATING OPERATION, *the reverse of the combining operation, is defined as follows. Starting from the root, each internal processor node sends a packet downwards to each of its children. The processor node (other than the root) sends a packet downwards after receiving another packet from its parent.*

Definition 4.1 defines two efficient operations, *combining* and *disseminating*, in a processor tree. The combining and disseminating operations are efficient because all the processors at the same tree level can be processed in parallel and therefore the latency is only the time spent in processor nodes along the critical path among all paths from leaves to the root. In addition, the total number of sends (or receives) is only about the number of processors. In the global

protocol, collecting processor status can be done via a combining operation, while broadcasting new values of π^{max} can be done via a disseminating operation.

4.1.2.2 Sparse Priority Distribution

In our model, load balancing becomes most complex when task priorities are *sparsely* distributed, i.e., when the total load of tasks with each priority is small. In this situation, if we still use the above simple global protocol to balance the load, the total load of tasks in T_0 (or T_1) is usually small. This implies that the GLB will soon need to update the value of π^{max}. That is, the GLB needs to set up new rounds (including broadcasting new π^{max} and collecting the load status) very often, resulting in excessive communication.

The key idea for solving this problem is to group additional highest-priority tasks into T_0 or T_1 at the beginning of each round so that the GLB will be able to donate more tasks from these sets and will not have to rebroadcast for the next round so quickly. Note that the technique of grouping tasks will result in schedules which do not strictly obey the order of priorities; but as mentioned in Chapter 2 the program correctness *should never rely on priorities*.

There is an important issue for grouping tasks: we do not want the total load of these grouped tasks to be too small or too large.

- If the total load is too small, the system will soon need to set up a new round again. Thus, the effect of grouping extra tasks does not help much in this case.
- If the total load is too large, the system may schedule many tasks with priorities lower than π^{max}; this fails to follow priority well. For example, for BFS, scheduling many tasks with low priority could end up wasting a significant amount of computation time.

So, the compromise is that for each round we want the total load of grouped tasks to be comparable to a load threshold $L_{thr} = cL_{ovh}$, where c is a large constant and L_{ovh} is the aggregate overhead associated with setting up a new round among all processors. With this compromise, we make the overhead (L_{ovh}) less significant while keeping the total load of extra grouped tasks as small as possible. Since setting up a new round requires a disseminating operation to broadcast the new value of π^{max} and a combining operation to collect all processors'

load status, the overhead L_{ovh} is[1] $O(p \log p)$ sends/receives, which can be roughly predicted *a priori*.

To be more specific about the compromise, we will, during each round, group the highest-priority tasks whose total load is $L_{grp} = \Theta(L_{thr})$, if the total load of tasks in the whole system is $L_{total} = \Omega(L_{thr})$ and each task's grain size is $O(L_{thr})$. If the total load of tasks is $L_{total} < L_{thr}$, then $L_{grp} = L_{total}$ because we can at most group tasks with total load L_{total}. If the maximum task grain size G_{max} is larger than L_{thr}, then $L_{grp} = O(G_{max})$ because we may need to group the task with G_{max} (e.g., when the task with the grain size G_{max} has the highest priority).

In Section 5.2, we will design an efficient algorithm, called the *parallel range selection* (*PRS*) algorithm, on a processor tree with a constant degree (≥ 2); this algorithm can solve the sparse priority distribution problem and satisfy the following two properties:

O1 The algorithm only requires one combining operation and then one disseminating operation.

O2 Each packet size is $O(\log^2 p)$.

In the above problem, we allow the total load of grouped tasks, L_{grp}, to be in a range $\Theta(L_{thr})$, not just a fixed value (say L_{thr}), for the following two reasons.

- It appears to be hard to implement an efficient algorithm to group the highest-priority tasks with a fixed total load L_{thr}, while satisfying Properties O1 and O2 simultaneously.
- It is not critical for L_{grp} to be exact because the environment is changing. During the period when we group tasks for T_0 or T_1, the load status on each processor may have, more or less, been changed due to new task scheduling and task creation.

4.1.2.3 Advanced Global Protocol

In this section, we will modify the simple global protocol by adding the PRS algorithm (to be described in Section 5.2) with properties O1 and O2, such that the protocol can also cope

[1] If $g(n) = O(f(n))$, there exists some positive c for which $g(n) \leq cf(n)$ for all sufficiently large n.
If $g(n) = \Omega(f(n))$, there exists some positive c for which $g(n) \geq cf(n)$ for all sufficiently large n.
If $g(n) = \Theta(f(n))$, then $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$.

with the problem of sparse priority distributions. In this advanced global protocol, the PRS algorithm is used to group a set of additional highest-priority tasks at the beginning of each load balancing round. These tasks are grouped into T_1 (i.e., grouped into the T_0 of the *next* load balancing round in advance) so that we can simultaneously perform the PRS operation and balance the tasks in the current T_0.

For the new design, the two sets T_0 and T_1 are changed as follows. We define $T_0 = T'_0 + T''_0$, where T'_0 is the set of tasks with priority π^{max} and T''_0 is the set of tasks determined by the PRS algorithm in the previous round. Similarly, we define $T_1 = T'_1 + T''_1$, where T'_1 is the set of tasks with priority $\pi^{max} - 1$ (excluding those tasks in T_0) and T''_1 is the set of tasks determined by the PRS algorithm in the current round. In fact, T'_0 and T'_1 are just T_0 and T_1 in the original global protocol.

The new protocol is the same as the original protocol except for the following. At the very beginning of a round, each processor moves tasks from T''_1 to T''_0, reports processor status, and starts doing the combining operation of the PRS algorithm. Since the status report also requires a combining operation, we can combine the two combining operations in order to reduce communication overhead. When the root of GLB receives all the reports, it will check whether the total load of tasks in T'_1 is already large enough (e.g. greater than L_{thr}). If so, we do not need to apply the PRS algorithm to group tasks for the next round. If not, we will do the disseminating operation of the PRS algorithm to group more highest-priority tasks into T''_1. This provides us with a chance to omit the disseminating operation when T'_1 is already large enough. We will show below that the *PDC-WK* scheduling algorithm can usually omit the disseminating operation.

Before examining the case of the *PDC-WK* scheduling algorithm, we need to carefully define the task grain size. We define the grain size of a task to be the average time between the moment the task begins executing and the moment the corresponding processor requests the next task from another processor, not just the amount of computation taken for the task. For example, for the *PDC-WK* scheduling algorithm the grain size of each task is quite large because we usually exhaust all the local tasks and their descendants before requesting a task from other processors. For *PBFS-GPQ*, the grain size of a task is about the amount of time for computing the task because the algorithm is very likely to schedule the next task with the highest priority from another processor. Although the definition of task grain size is not so straightforward, we argue that the scheduling programmer can help the application programmer

to calculate the task grain size.

Now, let us examine the case of the *PDC-WK* scheduling algorithm according to the above definition. Since each node's grain size is very large, the total load in T'_i tends to be very large as long as T'_i has one node. Consequently, the disseminating operation is usually omitted, and the protocol is almost the same as the original one in this case. Our experiments presented in Chapter 6 also confirm that the disseminating operation is usually omitted for *PDC-WK*.

4.2 Maintaining Physical Lists

Since PLs are similar to priority queues, PLs can be maintained in the same way as priority queues. A priority queue data structure often requires the following primitive operations.

INSERT($\mathcal{T}$,π): Inserts a task $\mathcal{T}$ with priority π into the priority queue.

DELETE($\mathcal{T}$): Deletes a task $\mathcal{T}$ from the priority queue. Note that in the multilist scheduling model if one task is scheduled from one PL, then we need to delete the instances of that task from other PLs on the same processor too.

MAXPRI(): Returns the highest priority from the queue.

DELETEMAX(): Deletes and returns the task with the highest priority in the priority queue.

As described in Section 2.2, since there may be some derived PL based on this PL with a monotonically *decreasing* priority translation function, we also need to provide MINPRI and DELETEMIN primitive operations. This is because the highest priority in the derived PL is the lowest priority in the base PL.

The above operations are called *DEQ* operations[2] when they access/insert/delete a task with the highest or lowest priority. (The MAXPRI, DELETEMAX, MINPRI, and DELETEMIN operations

[2] A "DEQ operation" is our preferred name for what others (e.g., [97]) have called a "deque operation", i.e., an operation on a "double ended queue", which is accessed only at its two ends. We avoid the term "deque operation" because this sounds like removal from a queue.

are always DEQ operations while the INSERT and DELETE operations are sometimes DEQ operations.)

For the PLs involved in a global scheduling subpattern, we may also need to provide other operations required by the PRS algorithm (described in Section 5.2). They include:

THRESHPRI(L_{thr}), where $L_{thr} \geq 0$: If $L(-\infty) < L_{thr}$, THRESHPRI returns $-\infty$ and $L(-\infty)$; otherwise, there is a priority π such that $L(\pi+1) < L_{thr} \leq L(\pi)$, and THRESHPRI returns π and $L(\pi)$, where $L(\pi) = \sum_{\mathcal{T}}(G_{\mathcal{T}})$ for all tasks $\mathcal{T}$ with priorities $\pi' \geq \pi$. Note that we can substitute $\pi_{min} - 1$ for $-\infty$, and $\pi_{max} + 1$ for ∞, where π_{min} (π_{max}) is the highest (lowest) priority which can be used. The PRS algorithm needs to use this operation to obtain the priority distribution.

SPLIT(π): Splits the priority queue into two, one part containing all the tasks with priority $\pi' \geq \pi$ and the other part containing all the other tasks. The PRS algorithm will need this operation to select all the tasks with priorities greater than or equal to a threshold priority.

Here, we should note that if a global scheduling subpattern uses a derived PL based on another PL and a priority translation function f, the derived PL needs to translate its priority threshold π to the priority in the base PL in order to perform the SPLIT operation. Since the function f only translates the priorities in the base PL to those in the derived PL, the programmer needs to provide the inverse of function f to translate priorities from the derived PL to those in the base PL. If function f is linear (as in the *PDC-WK* scheduling algorithm and factoring scheduling algorithm), i.e., $f(\pi) = a\pi + b$, the programmer only needs to specify the constants a and b; the system, knowing that f is linear, can automatically find the inverse of function f.

Now, we want to see how to support the above operations for derived PLs and base PLs. For a derived PL, we can perform all the PL operations based on the base PL and the priority translation function (maybe including the inverse of the function). For a base PL, we will, in Section 4.2.1, propose an efficient priority queue supporting the above operations. This proposed priority queue will satisfy the following two properties.

P1 The worst-case times of all the operations are $O(\log n)$, where n is the number of priorities in the priority queue. (Note that the worst-case time of the operation MAXPRI or MINPRI

is only $O(1)$.)

P2 The amortized times of all the DEQ operations is $O(1)$. The amortized time [92] is defined as the average time of an operation in a worst-case sequence of operations.

The first property P1 implies that these operations are efficient for the worst case. The worst-case time is important because it may shorten the response time for interprocessor communication operations. For example, assume that the worst-case time is $O(n)$. Even if the amortized time is much less than $O(n)$, the donation of a particular task may be significantly delayed due to some operation requiring $O(n)$ computation time. Therefore, the performance may become bad and unpredictable.

The second property P2 implies that the DEQ operations are optimal. This property is also important because the DEQ operations happen in many cases. As mentioned above, the operations, DELETEMAX, MAXPRI, DELETEMIN and MINPRI, are always DEQ operations. In addition, some applications tend to insert/delete a task only at the two ends. For example, in *PDC-WK*, when we schedule the deepest node locally, the node has the highest local priority, but has the lowest global priority; when we schedule the node closest to the root from all the (other) processors, the node has the highest global priority, but has the lowest local priority.

4.2.1 Data Structure of Priority Queue

Our priority queues are based on 2-3 trees [3], a kind of balanced search trees, which have the following two basic properties.

- Each internal node has 2 or 3 children except that the root may have less than two children in the case that there are less than two nodes in the whole priority queue.
- All leaves are at the same depth.

From the two above properties, for a 2-3 tree with height $h > 1$, the minimum number of nodes is 2^h and the maximum number of nodes is 3^h. Thus, $h = O(\log n)$, where n is the number of nodes in the tree.

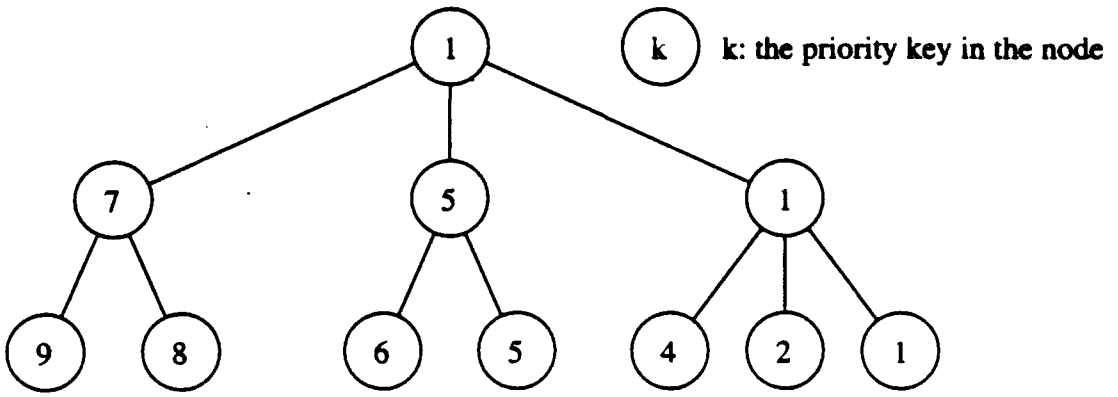


Figure 4.5: A 2-3 tree.

In 2-3 trees, each node is associated with a priority key; each leaf node is also associated with those tasks whose priorities are the same as the leaf's priority. These priorities are ordered as follows. For each internal node ν whose priority is denoted by π and whose ith child and ith child's priority are denoted by ν_i and π_i, there are two ordering restrictions for these priorities: (1) $\pi \leq \pi_i$ for all i; (2) if $i < j$, all the priorities of nodes in the subtree rooted at ν_i must be larger than those at ν_j (note that we say that ν_i is to the *left* of ν_j). An example is illustrated in Figure 4.5. Because of the second restriction, for a given node and a given priority, we can easily find the child of the node whose subtree may have the leaf with the priority. We can use this technique to search for a leaf whose priority is adjacent to a given priority. Since the priorities of leaves are all distinct, we store all tasks with the same priority in the same leaf.

In [3], Aho *et al.* proved that the times of the operations INSERT, DELETE, DELETEMAX, MAXPRI, DELETEMIN, MINPRI, and SPLIT are only $O(\log n)$ because each of these operations only traverses a path between the root and some leaf at most downwards once and upwards once. First, we search from the root to the leaf at which the operation of accessing, insertion, deletion, or splitting takes place. Note that the DELETE operation does not need to search from the root because we can let every task have a pointer to the leaf (with the same priority as the task's) directly. After finding the leaf ν, we perform the desired operation and rebalance the tree if necessary to ensure that it is still a 2-3 tree. This requires only one upward pass.

In order to support the THRESHPRI operation, we need to let each node ν in the tree have a load variable L_ν. (Note that this is similar to the augmented tree described in [28, Chapter 15].) If ν is a leaf, $L_\nu = \sum G_\mathcal{T}$ for all tasks $\mathcal{T}$ in ν; otherwise ν is an internal node and $L_\nu = \sum L_{\nu'}$ for all children ν' of ν. An example is illustrated in Figure 4.6, in which the number in the

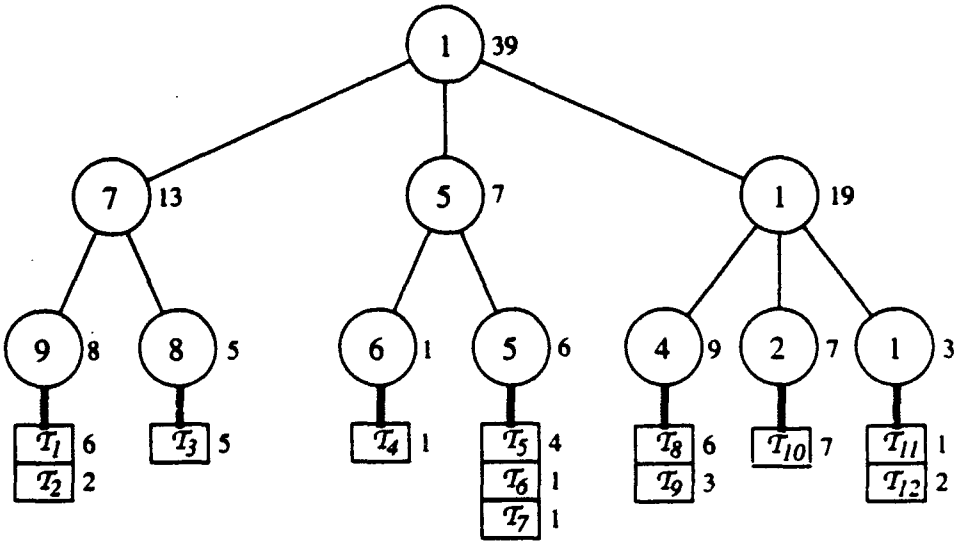


Figure 4.6: A 2-3 tree, showing load variable values.

right hand side of each rectangle (i.e., task) represents the grain size of the task, and the number in the right hand side of each circle (i.e., tree node) represents the load variable value of the node. So, whenever a leaf changes its load (e.g., a task is added to some leaf), we can just use the above formula to update the load variables L_ν from the leaf to the root. In addition, whenever reconfiguring the tree upwards, we may also need to calculate the load variables of these touched nodes again by the above formula. Thus, the other operations still need $O(\log n)$ time.

For the THRESHPRI operation with a given load threshold L_{thr}, if the total load is smaller than L_{thr}, we stop and return the total load and the threshold priority $-\infty$; otherwise, we will search along the path from the root to the leaf with the threshold priority satisfying the operation. For example, in Figure 4.6, if $L_{thr} = 45$, we will return the total load 39 and the threshold priority $-\infty$; if $L_{thr} = 15$, we will search from the root to the leaf with priority 5 and return the load 20 ($= 8 + 5 + 1 + 6$) and the priority 5. In the latter case, starting from the root, we repeatedly visit one node downwards until a leaf is reached such that for each visited node ν, the following condition holds: $L_l < L_{thr} \leq L_l + L_\nu$, where L_ν is the load variable of ν and L_l is the total load of tasks in all the subtrees to the left of ν (i.e., the total load of tasks with priorities $> \pi'$, where π' is the largest priority of the leaves in the subtree rooted at ν). It is trivial that the above condition holds for the root, initially. For example, in Figure 4.6, for the root, since $L_l = 0$ and $L_\nu = 39$, the above condition holds (with $L_{thr} = 15$). Then, when visiting the node ν on the path with the above condition, we can guarantee from the definition of

load variables that one of its children must satisfy this condition too. For example, for the root in Figure 4.6, we can find the internal node ν (with priority 5) satisfying the above condition (with $L_{thr} = 15$), since $L_l = 13$ and $L_\nu = 7$. The reader can verify that the condition holds for the leaf with priority 5, too. When the leaf ν with priority π is reached, we can derive the result $L(\pi + 1) < L_{thr} \leq L(\pi)$ since $L_l < L_{thr} \leq L_l + L_\nu$, $L_l + L_\nu = L(\pi)$, and $L_l = L(\pi + 1)$. So, π is just the threshold priority for the function. From the above, the THRESHPRI operation only needs to visit nodes $O(\log n)$ times.

Amortized Time

We will further modify the data structure such that the amortized times for DEQ operations satisfy Property P2. For insertion/deletion, many researchers [47, 68] have proved that the amortized time for rebalancing the tree is only $O(1)$. However, DEQ operations may still need $O(\log n)$ time for the following two basic procedures: (1) to search downwards to the leaf at which the access, insertion, or deletion takes place and (2) to update the load variables upwards. So, in order to let the times for the DEQ operations still be amortized $O(1)$, we will modify the data structure to reduce the computation times of the above two procedures for DEQ operations to $O(1)$, as follows.

First, we put a *finger* at each of the two ends; i.e., the priority queue has two special fingers pointing to the leaves with the highest priority and the lowest priority. (Note that putting fingers on a search tree [38, 98] is a common technique for finding special nodes and their neighbors more quickly.) Hence, for DEQ operations, we can directly find the leaf from the two fingers without searching downwards from the root to the leaf. Thus, for each DEQ operation, the time for searching the leaf is $O(1)$. In addition, with the same technique, the total worst-case times of the MAXPRI and MINPRI operations are only $O(1)$.

Second, we allow that nodes on the *leftmost* and *rightmost paths* do not need to have accurate load status, where the leftmost (rightmost) path is the path between the root and the leaf with the lowest (highest) priority. So, we do not need to update the load variables of the nodes on both paths, as shown in Figure 4.7. Thus, for all the DEQ operations, we do not have to update the load upwards, i.e., the computation time for updating the load is $O(1)$. With this modification, each THRESHPRI operation needs to update the load status of nodes on both paths (upwards) first and then perform the original THRESHPRI operation. Thus, the new THRESHPRI

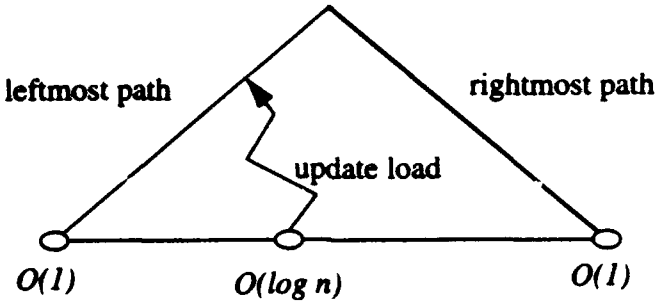


Figure 4.7: Updating load variables.

operation still needs $O(\log n)$ time.

4.3 Discussion

Although we have proposed some efficient techniques to implement the multilist model, it is still very difficult to argue that our model achieves the ultimate goal: for *all* scheduling algorithms, our general approach incurs no significant overhead. So, we will leave this problem open and only argue that our general approach incurs no significant performance overhead at least for the following four important scheduling algorithms: *PBFS-GPQ*, *PBFS-R*, *PDC-WK*, and *PDC-RR*.

1. For the *PBFS-GPQ* scheduling algorithm, researchers in [76] also used a centralized mechanism to maintain the global priority queue. They did not consider the sparse priority distribution situation because their task grain size was coarse enough and they had relatively few processors. But, if their task grain size were small or if they had many processors, we expect that our advanced global protocol could be used to solve their problem. So, if a dedicated design for *PBFS-GPQ* followed our protocol, our general system should perform almost as efficiently as the dedicated one.

2. For the *PBFS-R* scheduling algorithm, Karp and Zhang [57] used a randomized technique. The corresponding scheduling scheme based on our model (as shown in Section 3.1.1.2) merges PLs into local VLs. Because of this locality, our model will perform almost as efficiently as a dedicated design for *PBFS-R*.

3. For the *PDC-WK* scheduling algorithm, Wu and Kung [105] used a global pool to contain all the nodes at the highest available tree level in the D&C computation tree. Whenever some processor is idle, nodes in the pool can be donated to the idle processor. In fact, this algorithm is the same as the simple global protocol except for using T_1 (see Section 4.1 2.1). Note that the set T_0 is equivalent to the global pool. In our system, the set T_1 is used to make the transition from the current maximum priority to the next maximum priority smoother. In fact, a dedicated design for parallel D&C may also benefit from using this technique of adding the set T_1.

   In the advanced protocol, we may require an extra combining and disseminating operation when there is a sparse priority distribution. However, for *PDC-WK*, we can assign a very large grain size to each node, such that the total load of tasks in T_1' is very large in most cases. Since T_1 already has enough load, we do not need to use the extra disseminating operation to group extra tasks into T_1''. Thus, the amount of communication is almost the same as the simple global protocol or a dedicated *PDC-WK* algorithm.

   In addition, for PLs, the *PDC-WK* algorithm always inserts and deletes a task with the highest priority or the lowest priority, as mentioned earlier. Thus, in our system, the amortized times for these DEQ operations are only $O(1)$. In a dedicated *PDC-WK* design, we can use a doubly-linked list to maintain these tasks such that the time for each DEQ operation is only $O(1)$. Our system may still not be as good as the dedicated design which uses a doubly-linked list, but it is only within a constant factor. In the future, we may even provide several types of data structures (including doubly-linked list) and allow programmers to choose their preferred data structures. From the above, we can conclude that our general system incurs no significant performance overhead for *PDC-WK*.

4. For the *PDC-RR* scheduling algorithm in Section 3.1.2.2, researchers in [34, 82, 44] all used the following strategy: an idle processor requests a task from another processor in a round-robin fashion. Section 4.1.1 showed that our system can judge from the priority range information that a pre-selected processor does not need to report the status of its GL to its destination processors, unless its destination processors become idle and explicitly send task requests. Thus, the algorithm based on our model will perform almost as efficiently as a dedicated *PDC-RR* design.

The above shows that our system, based on a uniform scheduling model, can efficiently implement the above scheduling algorithms. In the past, it has been difficult for any single

scheduling system to efficiently support the scheduling algorithms for both parallel BFS and D&C. For example, Manber and Finkel in [34] provided a parallel programming system for both problems; however, they also pointed out that it is difficult for them to use a uniform framework to efficiently support these algorithms. We believe that our system is the first system that can do so.

Chapter 5

Selected Theoretical Topics

In order to develop our multilist scheduling system, we have also studied two theoretical topics. This chapter will present their results. First, Section 5.1 will present the communication complexity for parallel divide-and-conquer (D&C). The theoretical results in Section 5.1 were previously published in [105]. Second, Section 5.2 will propose an efficient algorithm for the parallel range selection problem that will be used in the implementation of our model (see Section 4.1.2.2). Since the two topics in this chapter have no strong relation to each other or to the rest of this thesis, the reader can skip this chapter without loss of continuity.

5.1 Communication Complexity for Parallel D&C

In this section, we will theoretically study the relationship between load balancing and communication cost for performing D&C computations on a parallel system. As described in Section 3.1.2, D&C is a common computation paradigm, in which the solution to a problem is obtained by solving subproblems recursively. Each node in the tree corresponds to a problem instance, and children of the node correspond to its subproblems. During the computation, each internal (non-leaf) node goes through two phases. The first phase is the *divide phase* during which the problem instance associated with the node is divided into subproblems. The second phase is the *combine phase* during which the solution of the problem instance associated with the node is derived by combining solutions of the subproblems associated with the node's children. After

its creation each leaf will perform some computation and return the results to its parent. At a given time, nodes on a wavefront that cuts across all paths from the root to leaves can be active in performing divide, combine, or compute operations. Along each path the wavefront first moves down from the root to its leaf and then up from the leaf to the root.

At first glance, one might think that it should be straightforward to perform D&C in parallel, because nodes on the wavefront can all be processed independently. However, if one wants to achieve good load balancing between the processors, then parallelizing D&C becomes nontrivial. In fact, doing efficient D&C on any real parallel machine has been a major challenge to researchers [33, 34, 82, 89] for many years.

The difficulties are due to the fact that many D&C computations are highly dynamic in the sense that these computations are data-dependent. During computation, a problem instance can be expanded into any number of subproblems depending on the data that have been computed so far. In fact, the trees of many D&C computations can be expected to be sparse and irregular, and as a result, load balancing must be adaptive to the tree structure and must be done dynamically at run time. This implies that computation loads need to be moved around between processors during computation. The challenge is then to devise efficient scheduling algorithms which can achieve good load balancing while minimizing the communication cost for moving computations around.

In general there is a tradeoff between balancing computation loads and minimizing communication costs. The results of this section quantify this tradeoff. In particular, this section establishes lower bounds on the communication cost for any scheduling algorithm based on how well it performs load balancing.

5.1.1 Summary of Results

5.1.1.1 Definitions and Notation

The tree of a D&C computation is called a (N, h, d)-*tree*, if

- N is the number of nodes in the tree,

- h is the height of the tree, and
- d is the maximal number of children of a node. (We assume that d is at least 2, to allow parallel processing of the tree.)

A node is said to be at tree level i if it is the i-th node on the path from the root to the node. Therefore, the root is at level 1, and the height of the tree is the maximal level number.

For the parallel system which will carry out the D&C computation, we assume that

- p is the number of processors in the system, and
- it takes one time step for a processor to *expand* a node, i.e., to perform the divide operation for an internal node, or to perform the compute operation for a leaf node. For simplicity, we assume that a processor takes no time to perform a combine operation.

When a node is expanded, zero or more children may be *generated*. More precisely, if a node does not generate any children, the node is a leaf; if a node generates one or more (up to d) children, the node is an internal node. Each newly generated node will in turn be expanded by some processor in the future. A *frontier node* is a node which has been generated but has not been expanded.

A *scheduling algorithm* for a D&C computation schedules nodes (i.e., frontier nodes) on processors for expansion. We assume that scheduling algorithms cannot "lookahead". This non-lookahead assumption is reasonable when dealing with irregular D&C trees. In this type of tree, the number of children a parent may have (if any) is typically data-dependent and is therefore not known *a priori*.

The *parallel computation cost* $T_{\mathcal{A}}(H)$ of a scheduling algorithm $\mathcal{A}$ for a D&C computation tree H is the maximum number of the nodes that any processor may expand. Since there are N nodes and p processors, a lower bound on $T_{\mathcal{A}}(H)$ is $T_{min} = \lceil N/p \rceil$. The parallel computation cost $T_{\mathcal{A}}$ of algorithm $\mathcal{A}$ is defined as the maximum $T_{\mathcal{A}}(H)$ for all (N, h, d)-trees H.

The *communication cost* $C_{\mathcal{A}}(H)$ of a scheduling algorithm $\mathcal{A}$ for a D&C computation tree H is the total number of cross nodes. A *cross node* is a node which is generated by one

processor but expanded by another processor. Note that the processor expanding a cross node needs to receive information from the processor generating the node. Therefore, $C_{\mathcal{A}}(H)$ is a reasonable measure for capturing the interprocessor communication cost in performing the divide phase of all the internal nodes. (A similar definition of communication cost is used by Papadimitriou and Ullman in [74].) The communication cost $C_{\mathcal{A}}$ of algorithm $\mathcal{A}$ is defined as the maximum $C_{\mathcal{A}}(H)$ for all (N, h, d)-trees H.

5.1.1.2 Main Results

Theorem 5.1 *For each scheduling algorithm $\mathcal{A}$ for a parallel system of p processors, for each integer p', $0 < p' \leq p$, and for each N, h, and d with the following two restrictions,*

S1. $N > 3pd^2h$, *and*

S2. $h > \lceil \log_d N \rceil + \lceil \log_d pdh \rceil + 1$,

there exists some (N, h, d)-tree H for which at least one of the following two properties is true:

Q1. *the parallel computation cost of the algorithm is $T_{\mathcal{A}}(H) \geq N'/p'$;*

Q2. *the communication cost of the algorithm is $C_{\mathcal{A}}(H) \geq C'$,*

where $N' = N - 3pd^2h$, $C' = p'\kappa$, $\kappa = (d-1)h'$, and $h' = h - \lceil \log_d N \rceil - \lceil \log_d pdh \rceil - 1$.

Many D&C computations are expected to satisfy restrictions S1 and S2. Since N is usually an exponential function of h, restriction S1 is easily satisfied in these cases. Restriction S2 roughly requires that $N < d^{h-2}/ph$. If a tree is perfectly balanced and each node has exactly d children, then N would be $\Theta(d^{h-1})$ instead. A perfectly balanced tree is easy for load balancing because the subtrees of each node have the same computation load. Restrictions S1 and S2 basically capture those interesting D&C computations with irregular trees. This class of D&C computations are exactly those for which one finds it difficult to achieve good load balancing without paying much in communication overheads. The lower bound on $C_{\mathcal{A}}(H)$, stated in Q2 of the theorem, provides an explanation of why this must be the case.

The two properties Q1 and Q2 in Theorem 5.1 can be expressed in terms of the quantities N, h, d (associated with the D&C tree) and p (associated with the parallel system) as follows. One can check that $N' \geq (1-\epsilon_N)N$ and $h' \geq (1-\epsilon_h)h$ for each positive $\epsilon_N \leq 1$ and $\epsilon_h \leq 1$, provided that $h \geq \frac{2\log_d ph+\log_d 3+6-\log_d \epsilon_N}{\epsilon_h}$ and $\frac{d^{\epsilon_h h}}{pd^4h} \geq N \geq \frac{3pd^2h}{\epsilon_N}$. (Note: if $h \geq \frac{2\log_d ph+\log_d 3+6-\log_d \epsilon_N}{\epsilon_h}$, then $\frac{d^{\epsilon_h h}}{pd^4h} \geq \frac{3pd^2h}{\epsilon_N}$; if $\frac{d^{\epsilon_h h}}{pd^4h} \geq N$, then $\epsilon_h h \geq \log_d N+\log_d pdh+3 \geq \lceil \log_d N \rceil + \lceil \log_d pdh \rceil + 1 = h - h'$, i.e., $h' \geq (1-\epsilon_h)h$; if $N \geq \frac{3pd^2h}{\epsilon_N}$, then $N' \geq (1-\epsilon_N)N$.) From this and the fact that $N' < N$ and $h' < h$, we note that N' and h' approach N and h respectively, when both ϵ_N and ϵ_h approach 0. Therefore, Q1 and Q2 in Theorem 5.1 become $T_{\mathcal{A}}(H) = \Omega(N/p)$ and $C_{\mathcal{A}}(H) = \Omega(pdh)$ for large h, when p' is close to p. Furthermore, we can slightly change the theorem as Corollary 5.2.

Corollary 5.2 *For each scheduling algorithm for a parallel system of p processors, for each positive $\epsilon_C < 1$, which can be arbitrarily close to 0, there are values of N, h, d, p, and $\epsilon_T(>0)$, for which if the parallel computation cost is between $\frac{N}{p}$ and $(1+\epsilon_T)\frac{N}{p}$, then the communication cost must be at least $(1-\epsilon_C)C_u$, where $C_u = pdh$.*

Proof. Let $p \geq \frac{3}{\epsilon_C}$ and $d \geq \frac{3}{\epsilon_C}$. Then, let $\epsilon_T = \frac{1}{2p}$. And, let N and h be in the range as shown above with $\epsilon_h = \frac{\epsilon_C}{3}$ and $\epsilon_N = \frac{1}{2p}$. One can check that $(1+\epsilon_T)\frac{N}{p} \leq \frac{(1+\epsilon_T)N'}{p(1-\epsilon_N)} = \frac{N'(2p+1)}{p(2p-1)} < \frac{N'}{p-1}$ and $p'(d-1)h' \geq (1-\frac{1}{p})(1-\frac{1}{d})(1-\epsilon_h)C_u \geq (1-\frac{\epsilon_C}{3})^3 C_u > (1-\epsilon_C)C_u$ when $p' = p-1$. Thus, if $\frac{N}{p} \leq T_{\mathcal{A}} \leq (1+\epsilon_T)\frac{N}{p}$ $(< \frac{N'}{p-1})$, the communication cost must be at least $(1-\epsilon_C)C_u$. □

Theorem 5.1 also implies an important tradeoff result: if a scheduling algorithm wants to achieve a good load balancing by parallel processing, then it must pay a high price in communication cost. We can express the tradeoff between $T_{\mathcal{A}}$ and $C_{\mathcal{A}}$ explicitly by showing a lower bound on their product: $T_{\mathcal{A}} \cdot (C_{\mathcal{A}} + \kappa)$. If $(p^*-1)\kappa \leq C_{\mathcal{A}} < p^*\kappa$, where $0 < p^* \leq p$, then by Theorem 5.1, $T_{\mathcal{A}}$ must be at least N'/p^*. Therefore, $T_{\mathcal{A}} \cdot (C_{\mathcal{A}} + \kappa) \geq (N'/p^*) \cdot p^*\kappa = N' \cdot \kappa$. Note that because of $T_{\mathcal{A}} \geq N/p \geq N'/p$ this tradeoff is also satisfied when $C_{\mathcal{A}} \geq p\kappa$. This tradeoff result is summarized in Corollary 5.3 below.

Corollary 5.3 *For any scheduling algorithm $\mathcal{A}$ for a parallel system of p processors, for all N, h, and d with restrictions S1 and S2 as defined in Theorem 5.1,*

$$T_{\mathcal{A}} \cdot (C_{\mathcal{A}} + \kappa) \geq N' \cdot \kappa,$$

where N' and κ are defined in Theorem 5.1.

Theorem 5.4 *A scheduling algorithm $\mathcal{A}$ can be devised to have the property that the parallel computation cost is $T_{\mathcal{A}} = T_{min}$ and the communication cost is $C_{\mathcal{A}} \leq C_u (= pdh)$ for any (N, h, d)-tree.*

The algorithm satisfying Theorem 5.4 has the minimum parallel computation cost. By Corollary 5.2, the algorithm is optimal with respect to the communication cost, since the parallel computation cost of the algorithm is near optimal. These results also imply that the lower bound on $T_{\mathcal{A}} \cdot (C_{\mathcal{A}} + \kappa)$ in Corollary 5.3 is tight when both ϵ_N and ϵ_h are arbitrarily close to 0.

Note that Theorems 5.1 and 5.4 are so formulated that their results are *system-independent*. That is, the results are independent from the interconnection topology of the processors and various control overheads such as data structure maintenance and reading/writing messages. Therefore, our upper and lower bounds on $C_{\mathcal{A}}$ are intrinsic to any parallel system. These bounds give insights into actual communication cost in a real implementation, but exactly how they are related to the actual cost is a separate matter depending on the implementation. We have investigated this actual cost by implementing the algorithm on a variety of interconnection networks in [104].

Section 5.1.2 describes the algorithm of Theorem 5.4. Section 5.1.3 presents a simplified version of Theorem 5.1 and its proof to help the reading of Theorem 5.1. A complete proof of Theorem 5.1 is given in Section 5.1.4.

5.1.1.3 Relation to Past Work

There have been several approaches in performing parallel D&C. A simple approach (e.g., in [7]) is to expand all the nodes above a fixed level on one processor and then distribute nodes at this level to other processors. Load balancing would be done poorly in this approach when the tree is irregular. Another approach [89] is to distribute generated nodes, and to have each processor perform load balancing based on load status information from its neighbor processors. For this scheme, the communication cost can be very high in the worst case.

Recently, some researchers have made efforts to reduce communication overhead. A popular approach [34, 82, 108] is based on the "donate-highest-subtree" strategy, in which an idle processor will be given frontier nodes as near to the root as possible. Since a subtree rooted near the top usually has many nodes and these nodes can all be expanded locally, this strategy tends to reduce the amount of interprocessor communication. Ferguson and Korf [33] presented a D&C scheme with several processors scheduled first to a node and then to their children. The idea behind their scheme is also that of distributing frontier nodes near the root to idle processors.

Although the methods described in the previous paragraph all attempt to reduce communication overhead, they do not use global information to balance the load. It turns out that the communication cost for these methods can still be high in the worst case. For example, we estimate that the communication cost is $O(dh^{\log_d p})$ for Ferguson and Korf's scheme, and is $O(\min(p^2h, pdh^2))$ for the scheme in [34] with round-robin scheduling.

In contrast, the communication cost for the scheduling algorithm here (Section 5.1.2) is as low as $O(pdh)$ (Theorem 5.4). This is partly due to the fact that our algorithm is able to make effective use of global information (i.e., "global pool" in Section 5.1.2).

Most importantly, we note that none of the previous work has any lower bound results on the communication cost for parallel D&C computations. It appears that our lower bounds in Theorem 5.1 and Corollaries 5.2 and 5.3 are the first lower bound results for those D&C computations whose tree structures are dynamic in the sense that the tree structure is determined only at run time. Previous results on computation and communication cost tradeoffs such as those in [51, 52, 74] deal with only *static* computation graphs, whose topologies are known before the computation starts.

5.1.2 A Scheduling Algorithm and Upper Bounds

This section describes a new scheduling algorithm which can achieve the upper bounds in Theorem 5.4 for both parallel computation cost and communication cost. The bounds hold for any D&C computation, i.e., for any (N, h, d)-tree no matter how irregular it is.

Proposed Scheduling Algorithm

The scheduling algorithm uses a data structure, called a *Global Pool* (abbr. *GP*), to keep track of frontier nodes at a particular tree level which have not been taken by any processor for expansion. This level, identified by a variable gl, has the property that nodes at higher levels have all been taken by processors. Every processor will try to take a node from the GP to work on whenever it becomes idle. For the proof of Theorem 5.4, it suffices to assume that the GP is maintained by some single processor. (See [104] for a distributed scheme where the GP is maintained by multiple processors.)

Initially, the GP contains only the root and the value of gl is one. The GP becomes empty when all of its nodes at level gl have been taken by the processors. At this moment, all the processors are requested to send in their frontier nodes at level $gl + 1$ in the next time step when all the nodes at level $gl + 1$ have been generated. Then the GP is filled with this set of new nodes, and gl is increased by one. This process is repeated until all the nodes have been expanded.

The key idea of this algorithm is what each processor will do after it has taken a node from the GP. The processor will do a depth-first traversal. Consequently, the processor can exhaust all possible work locally before asking for a new node from the GP. As a result, we can prove (below) that the communication cost can be as low as C_u. While not related to parallel computation cost and communication cost, an important advantage of this local depth-first strategy is that it uses the minimum amount of memory.

In essence the scheduling algorithm described here uses a breadth-first scheme to distribute big chunks of computations to processors, and has each processor after receiving a computation follow the depth-first strategy locally. Therefore, the algorithm is a hybrid method, which interestingly will do a purely depth-first traversal of the tree in the case that only one processor

is used.

Suppose that we define the *parallel computation time* to be the time (in terms of number of time steps) when the last node is expanded by a processor. Then the parallel computation time of the algorithm described here is at most $\lceil N/p + h \rceil$. To see this, we note that some processors may become idle only when the number of nodes in the GP is smaller than the number of idle processors. In the worst case all the p processors may become idle at the end of some time step, but at this time there is only one node in the GP. Thus, in the next time step, as many as $p - 1$ processors may be idle. This situation can happen at most h times. Therefore, in the entire D&C computation, additional $h(p-1)$ nodes could have been expanded if there were no idle processors at any time step. This implies that the parallel computation time is at most $\lceil (N + h(p-1))/p \rceil \leq \lceil N/p + h \rceil$.

Note that parallel computation time defined in the previous paragraph is different from parallel computation cost defined in Section 5.1.1.1. Being able to take into account processor waiting time induced by inter-node dependency, parallel computation time may be of more practical interest than parallel computation cost.

However, to prove Theorem 5.4, we need to establish an upper bound on the parallel computation cost of the algorithm. We will do this and also establish an upper bound on the communication cost of the algorithm.

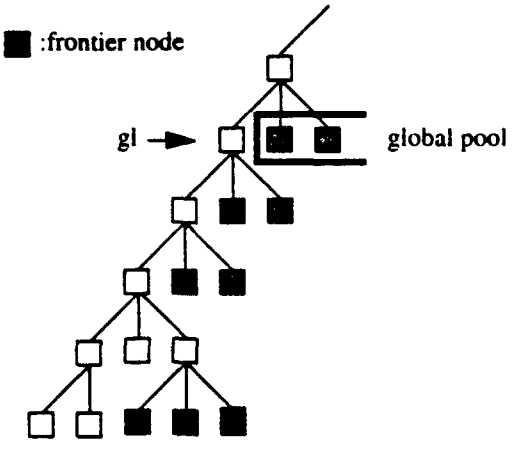


Figure 5.1: At most d frontier nodes at each level on a processor ($d = 3$).

Proof of Theorem 5.4. To achieve the $\lceil N/p \rceil$ upper bound on parallel computation

cost, we will need to add some fair scheduling feature to the algorithm described above. Whenever the number of nodes in the GP is smaller than the number of idle processors, we will select the active processors for the next time step from *all* the p processors in a fair way. That is, processors take turn to become active using a round-robin scheme. This ensures at the end of any time step that the total number of nodes expanded by a processor so far will not exceed that expanded by any other processor by more than one. Thus when all the N nodes are expanded, each processor will have expanded at most $\lceil N/p \rceil$. This proves that the parallel computation cost of the scheduling algorithm with the fair scheduling feature is at most $\lceil N/p \rceil$.

The communication cost of the algorithm is at most the number of frontier nodes entering the GP, as this represents the only interprocessor communication activity for the entire algorithm. Since by using depth-first search each processor has at most d local nodes at each level (as illustrated in Figure 5.1), the GP can collect at most pd nodes each time that gl increases. This will happen at most h times, so the total number of nodes entering the GP is bounded above by $C_u = pdh$.
□

Note that in a practical implementation, the fair scheduling feature may not be used since minimizing parallel computation cost may not be important. Without the fair scheduling feature, the parallel computation cost would become $\lceil N/p + h \rceil$. However, the communication cost can be reduced to $p(d-1)h$, if a processor right after expanding a node will schedule one child, if any, of the node for expansion at the next time step.

The scheduling algorithm described in this section is being used as a basis for developing a parallel programming model for D&C computations. To obtain practical insights, we plan to implement a programming system based on the model on the 26-host Nectar network system [6] developed at Carnegie Mellon University.

5.1.3 A Simplified Version of Theorem 5.1

This section presents Theorem 5.5 (see below), which is a simplified version of Theorem 5.1 dealing with only two processors. A relatively simple proof of Theorem 5.5 is given. This simple proof captures the essence of a more complicated proof of Theorem 5.1 given in Section 5.1.4. It is advised that the reader read this simple proof first to understand the ideas.

Theorem 5.5 *For each scheduling algorithm $\mathcal{A}$ for a parallel system of two processors, for each N, h, and d with the following three restrictions,*

S1. $N > 3dh$,

S2. $h > \lceil \log_d N \rceil + 2$, *and*

S3. $h - \lceil \log_d N \rceil - 2$ *is an even integer,*

there exists some (N, h, d)-tree H for which at least one of the following two properties is true:

T1. *the parallel computation cost of the algorithm is $T_{\mathcal{A}}(H) \geq N - 3dh$;*

T2. *the communication cost of the algorithm is $C_{\mathcal{A}}(H) \geq h'(d-1)$,*

where $h' = (h - \lceil \log_d N \rceil - 2)/2$.

Note that restrictions S1 and S2 correspond to those in Theorem 5.1. Restriction S3 is for a minor technical convenience, namely, ensuring that h' an integer.

Theorem 5.5 implies, for example, that if the communication cost is small (in the sense that T2 does not hold), then the parallel computation cost must be large (in the sense that T1 holds). In particular, if $C_{\mathcal{A}}(H) < h'(d-1)$ and if $3dh \ll N$, then the parallel computation cost will be close to N.

Proof of Theorem 5.5

Suppose that we are given a scheduling algorithm $\mathcal{A}$ for performing a D&C computation on processors P_1 and P_2. For algorithm $\mathcal{A}$, we will prove the existence of a (N, h, d)-tree H for which at least one of T1 and T2 must hold.

By playing an adversary game with algorithm $\mathcal{A}$, we will construct the tree by growing it from the root one step at a time. A time step consists of two phases, node scheduling phase and node expansion phase. In the node scheduling phase, algorithm $\mathcal{A}$ schedules a node or no node for each processor to execute. Then, in the node expansion phase, these scheduled nodes are expanded. In this phase we will determine the number of children each scheduled node will generate.

We will first define a special class of subtrees which will be used to describe some sufficient conditions under which a tree can grow to a (N, h, d)-tree. We will then give the main part of the proof including a description of the tree construction procedure.

HFD-Subtree

Definition 5.1 *At any given time during the tree construction, a High-and-Full-Degree subtree (abbv. HFD-subtree) is a subtree, which is rooted at a node at or above level $h - \lceil \log_d N \rceil$, and which has been constructed using the following rules:*

A1. *nodes above level h generate d children; and*

A2. *nodes at level h generate no children.*

Note that rules A1 and A2 imply that a node which is above level h and has no children must be a frontier node.

Lemma 5.1 *At any given time during the tree construction, if the current tree satisfies the following four properties:*

I1. *the total number of generated nodes is at most $N - h - d$ (generated nodes include the root);*

I2. *the height is at most h;*

I3. *the degree of any node is at most d; and*

I4. *the tree contains an HFD-subtree,*

then a construction procedure can be devised to grow the tree to a (N, h, d)-tree:

Proof. We first note that in the HFD-subtree of I4 there exist nodes which are above level h and have no children. Otherwise, the subtree would have been "fully grown" to level h, according to rules A1 and A2. Since its root is at and above level $h - \lceil \log_d N \rceil$, this fully grown HFD-subtree would have at least $d^{\lceil \log_d N \rceil} (\geq N)$ nodes. This contradicts I1. As noted above, those nodes in the current HFD-subtree which are above level h and have no children must all be frontier nodes.

Let H_1 be the current tree. We will identify a set of "padding nodes" which can be added to H_1 to make it a (N, h, d)-tree.

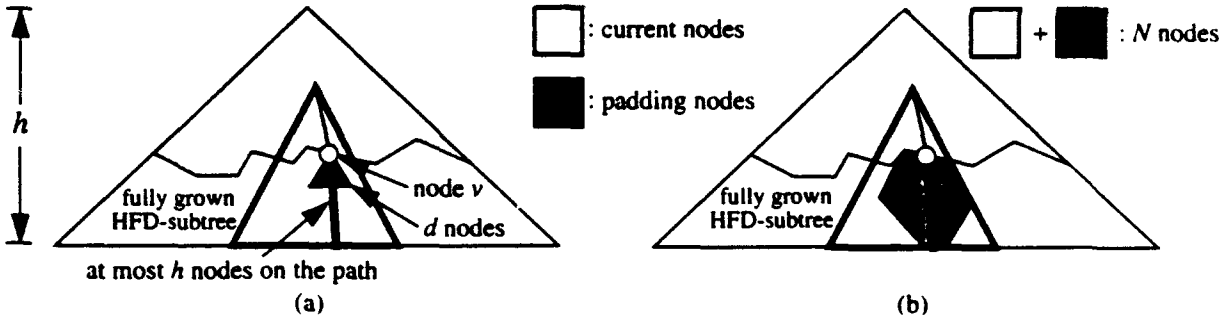


Figure 5.2: Growing the current tree to a (N, h, d)-tree.

If H_1 has height less than h or degree less than d, we will grow it by extending the current HFD-subtree from one of its frontier nodes which are above level h. Let v be this frontier node, as shown in Figure 5.2. We generate d children for v and create a path from v to a node at level h, as shown in Figure 5.2 (a). The resulting tree, called H_2, has height h, degree d, and no more than $(N - h - d) + d + h = N$ nodes.

If H_2 has less than N nodes, we will pad it with nodes in the fully grown HFD-subtree which are reachable from the current frontier nodes and other padding nodes, as illustrated in Figure 5.2 (b). Since the fully grown HFD-subtree has

at least N nodes, it has sufficient nodes which can be added to H_2 to make it a (N, h, d)-tree.

After having identified all these padding nodes, we now have a "blueprint" for a construction procedure to follow. More precisely, the construction procedure will just generate all those padding nodes in the dark region in Figure 5.2 (b). □

Main Part of Proof of Theorem 5.5

The tree construction procedure consists of three stages. Each stage uses an independent set of rules in constructing the tree.

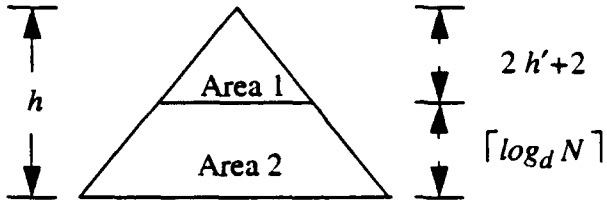


Figure 5.3: Two areas in the constructed tree.

In stage 1, we expand each node with exactly d children. Stage 1 terminates at time T_1 when a total of $2h'$ or $2h' + 1$ nodes have just been expanded. (Note that at this time the tree is completely inside area 1 of Figure 5.3.) Since the number of frontier nodes increases by $d - 1$ each time when a node is expanded, there are exactly $2h'(d - 1) + 1$ or $(2h' + 1)(d - 1) + 1$ frontier nodes at time T_1. Without loss of generality, we assume that processor P_1 has generated at least $h'(d - 1)$ frontier nodes.

Stage 2 starts right after T_1. In this stage every node above level h expanded by processor P_1 will have d children, while every node at level h or expanded by processor P_2 will have no children. Stage 2 terminates at time T_2 when one of the following two conditions becomes true:

C1 At least $h'(d - 1)$ cross nodes have been scheduled.

C2 At least $N - h - 2d$ nodes have been generated.

The following shows that C1 or C2 must become true sometime, i.e., T_2 exists. Recall that by the end of stage 1 processor P_1 has generated at least $h'(d-1)$ frontier nodes. In stage 2 processor P_1 will generate nodes in the subtrees rooted at those frontier nodes which are still in P_1. For each of these subtrees, since its root is in area 1 of Figure 5.3, the subtree can have at least $N-h-2d$ nodes unless some of these nodes are moved to processor P_2 from processor P_1. If C1 does not hold, then fewer than $h'(d-1)$ nodes can be moved from P_1 to P_2. Consequently, some subtree will have at least $N-h-2d$ nodes, and thus C2 will be true.

Stage 3 starts right after time T_2. Lemma 5.2 below shows that properties I1-I4 of Lemma 5.1 hold for the tree at time T_2. In stage 3, we follow the procedure described in the proof of Lemma 5.1 to grow the tree to a (N,h,d)-tree.

Lemma 5.2 *At any time in stage 1 or 2, including time T_2, the tree satisfies properties I1-I4 of Lemma 5.1.*

Proof. It is obvious from the descriptions of stages 1 and 2 that I2 and I3 are satisfied. For I1, we note that the total number of nodes generated in stage 1 is at most $(2h'+1)d+1$, and thus at most $N-h-d$ by restriction S1 of Theorem 5.5. In stage 2, I1 obviously holds when C2 is not true. Suppose that C2 becomes true at time T_2. Since the tree has no more than $N-h-2d$ nodes in the previous time step and since at most d nodes can be generated (in processor P_1) in one time step, there are at most $N-h-d$ nodes at time T_2.

Property I4 clearly holds for stage 1 by examining its description. It remains to prove that I4 holds for stage 2. The proof is similar to the earlier proof of the fact that C1 or C2 must become true in stage 2. Recall that in stage 1 processor P_1 has generated at least $h'(d-1)$ frontier nodes. We note that any of these subtrees rooted at these nodes is an HFD-subtree if the subtree does not contain any expanded cross node. Since the number of cross nodes expanded (not just scheduled) through time T_2 is less than $h'(d-1)$, one of these subtrees must be an HFD-subtree. Note that if C2 becomes true at time T_2 (in the node scheduling phase), the node scheduled has not been expanded. □

To complete the proof of Theorem 5.5, we observe that if C1 becomes true at some time in stage 2 or 3, it will remain true for the rest of the tree construction process. Therefore property T2 of Theorem 5.5 will hold for the final (N, h, d)-tree.

Now assuming that C1 never holds at any time in stage 2 or 3, we want to show that property T1 of Theorem 5.5 will hold for the final (N, h, d)-tree. We derive an upper bound on the total number of nodes expanded by processor P_2. The upper bound is the sum of four terms U_1, U_2, U_3 and U_4. In stage 1, processor P_2 has expanded at most $U_1 = 2h' + 1$ nodes. At time T_1, processor P_2 can have generated up to $(h' + 1)(d - 1) + 1$ frontier nodes, each of which can be expanded at most once by processor P_2 in stage 2 or 3. It is also possible for processor P_2 to expand nodes which are generated by P_1 but subsequently moved to P_2. The total number of these nodes is at most $C_{\mathcal{A}}(H) \leq U_3 = h'(d - 1)$. Moreover, to take care of the nodes generated after T_2 in stage 3, processor P_2 may expand up to $U_4 \leq h + 2d$ nodes. Therefore the total number of nodes expanded by processor P_2 is at most $U = U_1 + U_2 + U_3 + U_4 \leq 3dh$. This implies that processor P_1 has expanded at least $N - U = N - 3dh$; that is, property T1 holds. □

5.1.4 Proof of Theorem 5.1

Suppose that we are given a scheduling algorithm $\mathcal{A}$ for performing a D&C computation on a parallel system of p processors. For algorithm $\mathcal{A}$, we will prove the existence of a (N, h, d)-tree H for which either only p' processors are active for expanding most of nodes (at least N' nodes) or at least C' nodes are moved between processors to balance their computation loads. For the former, the parallel computation cost will be high, i.e., $T_{\mathcal{A}}(H) \geq N'/p'$ (property Q1). For the latter, the number of cross nodes will be large, i.e., $C_{\mathcal{A}}(H) \geq C'$ (property Q2).

By playing an adversary game with algorithm $\mathcal{A}$, we will construct the tree by growing it from the root one step at a time. The definition of time step is the same as that in the proof of Theorem 5.5.

We will give some more definitions in Section 5.1.4.1 and then give the main part of this proof in Section 5.1.4.2. All the related lemmas are in Section 5.1.4.3.

5.1.4.1 Definitions

To help derive a lower bound on the number of cross nodes, we introduce the following relation between subtrees.

Definition 5.2 *A set of subtrees is processor-or-ancestry independent (abbr. PA-independent) if for each pair of subtrees in the set at least one of the following two properties is satisfied:*

1. *Processor Independence: the roots of these two subtrees are generated on different processors;*
2. *Ancestry Independence: neither is a subtree of the other. That is, there is no ancestor-descendant relationship between the two roots.*

Note that for two PA-independent subtrees rooted at nodes r_1 and r_2, if node r_1 is an ancestor of node r_2, then both nodes must be generated on different processors. This implies that there must exist at least one cross node on the path from node r_1 (inclusive) to the parent (inclusive) of node r_2. Therefore, from this property, if there are k PA-independent subtrees each of which has at least one expanded cross node, then there are at least k expanded cross nodes in the tree. This is shown in Lemma 5.3 (in Section 5.1.4.3).

Definition 5.3 *An HFDC-subtree is an HFD-subtree (as defined in Definition 5.1) or a subtree with at least one cross node already expanded. If the root of an HFDC-subtree is generated on processor P, the subtree is called an HFDC-subtree on processor P.*

By Lemma 5.3 and Definition 5.3, if there are k PA-independent HFDC-subtrees and fewer than k expanded cross nodes, then there exists an HFD-subtree, as shown in Lemma 5.4. We will use this lemma to show the existence of an HFD-subtree during some periods of the tree construction procedure.

Stage 1 $\Longrightarrow$

Apply the following four rules:

R1. Nodes in area 1 (shown in Figure 5.5) will generate d children.

R2. Cross nodes in areas 2 and 3 (shown in Figure 5.5) will not generate any children.

R3. Non-cross nodes in areas 2 and 3 (excluding level h) will generate d children.

R4. Nodes at level h will not generate any children.

Repeat rules R1-R4 **until** time T_1 when any of the following three conditions holds:

C1. For some p' processors, at least h' non-cross nodes have been expanded on each processor.

C2. At least C' cross nodes have been scheduled.

C3. At least $N - (pd + d + h)$ nodes have been generated.

Stage 2 (continued from time T_1 when C1 holds) $\Longrightarrow$

Find a set Γ of p' processors with the following two properties:

B1. There are at least C' PA-independent HFDC-subtrees in Γ.

B2. There are at most h' non-cross nodes expanded on each of the other $p - p'$ processors in the set $\overline{\Gamma}$.

Apply the following three rules:

R5. Nodes (excluding those at level h) in Γ will generate d children.

R6. Nodes in $\overline{\Gamma}$ will not generate any children.

R7. Nodes at level h will not generate any children.

Repeat rules R5-R7 **until** time T_2 when either of the following two conditions holds:

C4. At least C' cross nodes have been scheduled.

C5. At least $N - (pd + d + h)$ nodes have been generated.

Stage 3 (continued from time T_1 when C2 or C3 holds or from time T_2 when C4 or C5 holds.) $\Longrightarrow$

Use the construction procedure described in the proof of Lemma 5.1 to grow the tree to a (N, h, d)-tree.

Figure 5.4: Tree construction procedure.

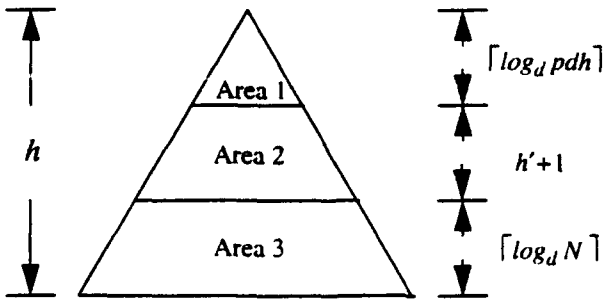


Figure 5.5: Three areas in the constructed tree.

5.1.4.2 Main Part of Proof of Theorem 5.1

The tree construction procedure consists of three stages. Basically, this procedure, summarized in Figure 5.4, is similar to that in Section 5.1.3. The main difference is that in stage 1 of this procedure we uses more sophisticated rules to prove a better lower bound of the number of cross nodes. (Note that if $h \gg \log_d N$, $p = 2$, and $p' = 1$, the lower bound of communication cost in this theorem is approximately twice as large as that in Theorem 5.5.)

In stage 1, we will repeatedly apply rules R1-R4 (in Figure 5.4) until time T_1 when one of the conditions C1-C3 holds. Rules R1-R4 ensure that each subtree rooted in area 1 or 2 is always an HFDC-subtree because in constructing the subtree either rules A1 and A2 are followed (using R1, R3, and R4) or some cross nodes are expanded (using R2). Basically, the procedure in stage 1 attempts to produce at least C' PA-independent HFDC-subtrees on some p' processors (property B1) while preventing each of the other $p - p'$ processors from expanding more than h' non-cross nodes (property B2). (Recall that in the proof of Theorem 5.5 subtrees rooted at frontier nodes at time T_1 are PA-independent HFDC-subtrees.)

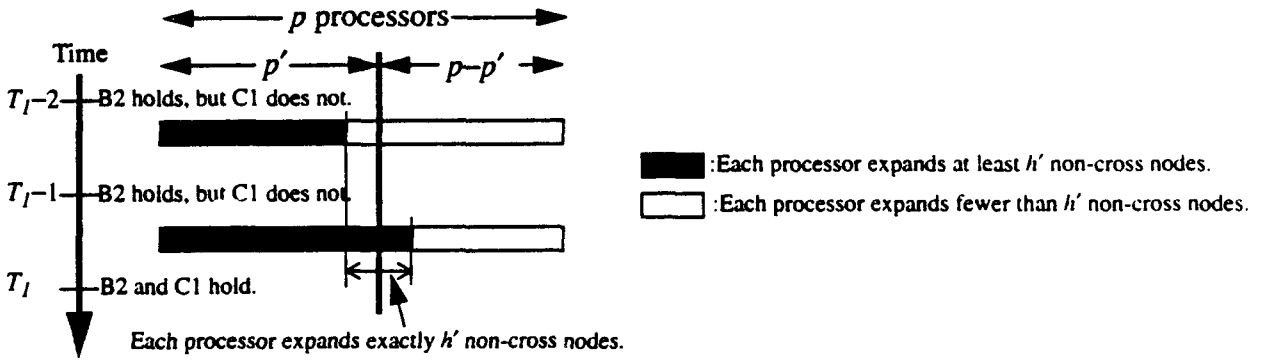


Figure 5.6: Around the time when condition C1 becomes true.

If condition C1 holds at time T_1, then from Figure 5.6 we can find a set Γ of p' processors for which condition C1 and property B2 hold. According to Lemma 5.5, there are at least $\kappa(= (d-1)h')$ PA-independent HFDC-subtrees on each processor which has expanded h' non-cross nodes. So there are at least $C'(= p'\kappa)$ PA-independent HFDC-subtrees in Γ at this time. Therefore, property B1 holds, and we are ready for stage 2.

In stage 2, we will repeatedly apply rules R5-R7 until time T_2 when condition C4 or C5 holds. (Note that these rules are exactly the same as those of stage 2 in Section 5.1.3.) According to property B1, initially, there are at least C' PA-independent HFDC-subtrees in Γ. In stage

2, these subtrees continue to be HFDC-subtrees, because either rules A1 and A2 are followed (using R5 and R7) or some cross nodes are expanded (using R6). In addition, by rule R6, the set $\overline{\Gamma}$ of the other $p - p'$ processors will not generate any new nodes.

Now, we want to show that one of the conditions C2-C5 must become true at time T_1 or T_2. According to Lemma 5.6 (in Section 5.1.4.3), at any time in stage 1 or 2 properties I1-I4 of Lemma 5.1 hold; so, at any time in stage 1 or 2 the tree will be able to grow to a (N, h, d)-tree by Lemma 5.1. Hence, if C2 or C4 never hold, C3 or C5 becomes true.

Stage 3 starts right after one of the conditions C2-C5 becomes true. (If C2 or C3 holds at T_1, this implies that stage 2 is empty.) Since Lemma 5.6 also shows that properties I1-I4 of Lemma 5.1 hold for the tree at time T_1 or T_2, in stage 3 we will follow the procedure described in the proof of Lemma 5.1 to grow the tree to a (N, h, d)-tree H.

To complete the proof, we observe that if $C_A(H) \geq C'$ it will remain true for the rest of the tree construction process. Therefore property Q2 of Theorem 5.5 will hold for H.

Now, assuming that $C_A(H) < C'$, we want to prove that property Q1 holds for H. Since C2 and C4 never hold, either C3 will become true at time T_1 or C5 will become true at time T_2. First, suppose that condition C5 becomes true at time T_2. To prove that property Q1 holds in this case, we will derive an upper bound on the total number of nodes expanded in $\overline{\Gamma}$. The upper bound consists of five terms U_1, U_2, U_3, U_4, and U_5. Assume that there are $C_1 < U_1 = C'$ cross nodes expanded in $\overline{\Gamma}$ in stage 1. In stage 1, the processors in $\overline{\Gamma}$ have expanded at most $U_2 = (p - p')h'$ non-cross nodes due to property B2. These nodes expanded in stage 1 will generate at most $U_3 = ((p - p')h' + C_1)d$ frontier nodes in $\overline{\Gamma}$ at time T_1, each of which can be expanded at most once in $\overline{\Gamma}$. After time T_1, it is also possible for the processors in $\overline{\Gamma}$ to expand nodes moved from the processors in Γ. The total number of these nodes is $U_4 \leq C_A(H) - C_1$. Moreover, to take care of the nodes generated after T_2, processors in $\overline{\Gamma}$ may expand up to $U_5 \leq pd + d + h$ nodes. Therefore, the total number of nodes expanded in $\overline{\Gamma}$ is at most $U = U_1 + U_2 + U_3 + U_4 + U_5 \leq 3pd^2h$. This implies that the processors in Γ have expanded at least $N - U = N - 3pd^2h$ nodes; therefore, $T_A(H) \geq (N - 3pd^2h)/p' \geq N'/p'$, i.e., property Q1 holds.

Suppose that condition C3 becomes true at time T_1. Since condition C1 does not hold in stage 1, we can find a set Γ of p' processors with property B2 (see Figure 5.6 also). Since stage

2 is empty for this case, we can let time T_2 be the same as T_1. Thus, we can use the same technique as above to prove that property Q1 holds. □

5.1.4.3 Relevant Lemmas

Lemma 5.3 *Suppose that there are k PA-independent subtrees at some time during the computation. If each of these subtrees has at least one expanded cross node, then the total number of expanded cross nodes in the whole tree constructed so far is at least k.*

Proof. This proof is not trivial because among these subtrees those with ancestry relationship may contain the same expanded cross node.

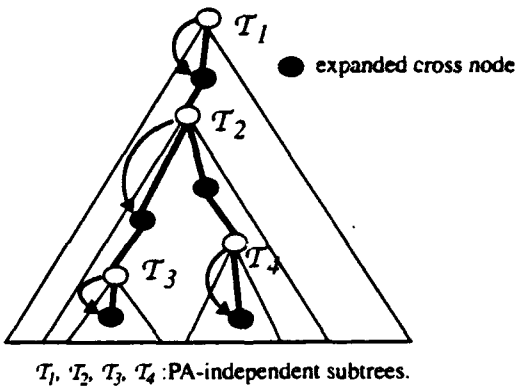


Figure 5.7: Expanded cross nodes corresponding to PA-independent subtrees.

In this proof, we will prune the k PA-independent subtrees one by one under the restriction that the subtree being pruned contains no other subtrees which have not been pruned yet. (For the example illustrated in Figure 5.7, we can prune the subtrees in the order: $\mathcal{T}_4$, $\mathcal{T}_3$, $\mathcal{T}_2$, and $\mathcal{T}_1$.) For this proof, it suffices to prove that each pruned subtree has at least one expanded cross node.

Initially, the first pruned subtree obviously has at least one expanded cross node by the assumption of the lemma. As mentioned in Section 5.1.4.1, for any two PA-independent subtrees $\mathcal{T}$ and $\mathcal{T}'$ rooted at nodes r and r' respectively, if r is an ancestor of r', there must exist at least one expanded cross node on the path from r

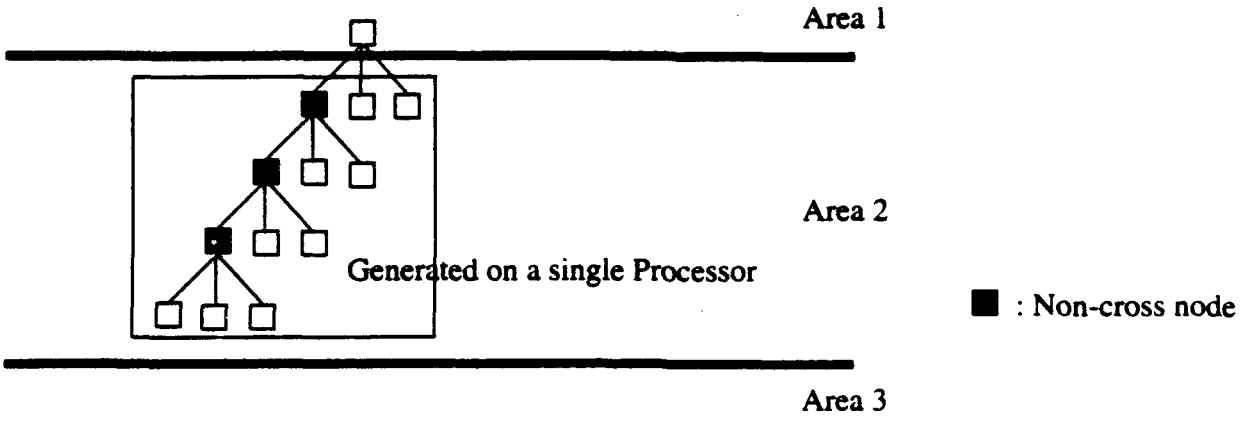


Figure 5.8: In stage 1, any non-cross node's ancestors in area 2 must have been generated on the same processor.

(inclusive) to the parent (inclusive) of r' due to processor independence. Therefore, if we prune $\mathcal{T}'$ at r', $\mathcal{T}$ still has at least one expanded cross node. Hence, after we prune each subtree under the above restriction, each of the remaining subtrees will still have at least one expanded cross node. This implies that the next pruned subtree also has at least one expanded cross node. So, each pruned subtree has at least one expanded cross node. □

Lemma 5.4 *At some time, if there are k PA-independent HFDC-subtrees and fewer than k expanded cross nodes, there exists an HFD-subtree.*

Proof. Assume that there exists no HFD-subtree. Thus, each of these PA-independent HFDC-subtrees has at least one expanded cross node according to the definition of HFDC-subtree. By Lemma 5.3, there are at least k expanded cross nodes. This is contradictory to the assumption of the lemma. □

Lemma 5.5 *In stage 1, if a processor has expanded h' non-cross nodes, then there are at least κ PA-independent HFDC-subtrees on the processor.*

Proof. As mentioned in Section 5.1.4.2, each subtree rooted in area 1 or 2 is always an HFDC-subtree in stage 1. Thus it suffices to prove that at least κ nodes with ancestry independence in areas 1 and 2 will be generated on the processor after h' non-cross nodes have been expanded. By rules R1-R3, for any non-cross node, all of its ancestors in area 2 (with $h'+1$ levels) must be non-cross nodes as shown in Figure 5.8. So, all the nodes generated by the first h' non-cross nodes must be in areas 1 and 2. Since each of the h' non-cross nodes will generate d children and can remove at most one ancestor, these non-cross nodes will, in total, generate at least $(d-1)h'(=\kappa)$ nodes with ancestry independence. □

Lemma 5.6 *At any time in stage 1 or 2, including time T_1 or T_2, the tree satisfies properties I1-I4 of Lemma 5.1.*

Proof. It is obvious from rules R1-R7 that I2 and I3 are satisfied. In addition, it is also obvious that I1 holds before condition C3 or C5 becomes true. Consider the first time step when at least $N-(pd+h+d)$ nodes have been generated (i.e., condition C3 or C5 holds). Since the tree has no more than $N-(pd+h+d)$ nodes in the previous time step and since at most pd nodes will be generated in each time step, there are at most $N-h-d$ nodes in the current time step. In the rest of this proof, we will show that I4 always holds (i.e., there always exists an HFD subtree) in each stage.

In stage 1, all the nodes in area 1 will generate d nodes by rule R1. So, before all the nodes in area 1 have been expanded, there must exist one frontier node in area 1, of which the subtree (with only one node) is an HFD-subtree. After all the nodes in area 1 are expanded, there are at least $d^{\lceil \log_d pdh \rceil} \geq pdh \geq C''$ subtrees rooted at the top level of area 2. Obviously, these subtrees are PA-independent. They are also HFDC-subtrees because each subtree rooted in area 1 or 2 in stage 1 is always an HFDC-subtree as described in Section 5.1.4.2. Since the number of expanded cross nodes is always less than C' (due to condition C2), there has always been an HFD-subtree up to time T_1 by Lemma 5.4. Thus, we can conclude that there always exists an HFD-subtree in stage 1.

In stage 2, initially, there are at least C' PA-independent HFDC-subtrees in Γ (property B1). These subtrees will continue to be HFDC-subtrees in this stage

as described in Section 5.1.4.2. In stage 2, due to condition C4 the number of expanded cross nodes is always less than C'; so, there always exists an HFD-subtree by Lemma 5.4. □

5.2 Parallel Range Selection

Selection [17, 28, 42] is a very common operation, which we define as follows.

> Given a set of N elements each containing a key, and given an integer value M, $1 \leq M \leq N$, select M elements that have the smallest key values[1].

(Note that when applied to our multilist scheduling system, elements are equivalent to tasks and keys are equivalent to priorities.)

For the parallel selection problem, some efficient algorithms have been designed by other researchers [16, 77], but they usually do not try to minimize the number of critical-path sends/receives[2]. For example, we estimate that the parallel selection algorithm [16] (a straightforward parallel design of [42]) requires an average of $O(\log p \log N)$ critical-path sends/receives. However, in a network-based multicomputer, we want to minimize the number of critical-path sends/receives while using moderately large packets, because a network-based multicomputer has the following characteristics. First, each send/receive incurs a significant amount of overhead, e.g., a couple of milliseconds over Ethernet or about 200 microseconds [27] on Nectar, as opposed to tens of nanoseconds per instruction. Second, a packet with moderate size does not incur a significant amount of overhead. For example, sending a packet with a few kilobytes is only a small number of times longer than that for a single word.

If we wanted to reduce the number of critical-path send/receives to $O(\log p)$ while using

[1] Sometimes, the definition of selection is to select only the element with the M-th smallest key. However, a selection algorithm which can find the M-th smallest key π usually can identify the M elements with priorities $\leq \pi$.

[2] Here, we view a "path" as a sequence of sends/receives performed on any corresponding sequence of processors. The path with the largest number of sends/receives will be called the critical path. Sends/receives on the critical path are called critical-path sends/receives. The time for a computation is at least the number of critical-path sends/receives times the average time for each send/receive.

packets of *unlimited* size, we could use a naive algorithm in which each processor sends a packet of M smallest-key values in order to construct the set of M smallest key values of the entire system. Since M is independent of the number of processors and can be an extremely large number, this algorithm may have poor performance. But if we limited the packet size to a moderate size (say, a polynomial function of $\log p$), it would be very difficult to reduce the number of critical-path sends/receives to $O(\log p)$.

Fortunately, as explained in Section 4.1.2.2, in our multilist scheduling system, it is not critical for the selection problem to select exactly M elements. It is good enough to select $\Theta(M)$ elements. Thus, we can relax the selection problem to the following:

> Given a set of N elements each containing a key, and given an integer value M, $1 \leq M \leq N$, select N_{sel} elements that have the smallest key values, where $N_{sel} = \Theta(M)$, e.g., $M \leq N_{sel} \leq 2M$.

This problem is called the range selection problem because the value N_{sel} is in a range $\Theta(M)$, not just a fixed value, M.

By taking advantage of this relaxation, we can devise a very efficient *parallel range selection* (*PRS*) algorithm which minimizes the number of critical-path sends/receives (by using only one combining operation and then one disseminating operation, as defined in Definition 4.1), while keeping the packet size moderate ($O(\log^2 p)$). First, we will summarize our theoretical results concerning the PRS algorithm, and then we will present more details.

5.2.1 Summary of Results

5.2.1.1 Assumptions

Our new PRS algorithm is based on a tree-shaped network of processors. For simplicity of discussion, we make two assumptions about the network as follows.

- The processor tree is a complete binary tree. This implies that the leaf processors are all at the same bottom level q of the processor tree, where q is $\lg p$ and p is the number of

leaf processors.

- All elements are distributed over leaf processors, not over internal processor nodes. For example, if internal processor nodes are embedded in leaf processors, as in the example in Figure 4.4, we can say the internal processor nodes have no elements.

Without loss of generality, we also assume all the elements have distinct key values for the range selection problem. If elements are allowed to have the same key values, we can redefine the key value in order to make each element have a distinct key value, as follows. For an element on a leaf processor P_i, if the original key value is π and the element is the jth element with the key value of π on P_i, we can use a compound key (π, i, j) as its new key: π is the primary key, i is the secondary key, and j is the tertiary key.

5.2.1.2 Main Result

Theorem 5.6 *Given a processor tree with the above assumptions, an algorithm can be devised to solve the PRS problem and to satisfy the following properties.*

U1 *The algorithm only requires one combining operation and then one disseminating operation (see Definition 4.1).*

U2 *Each packet size is* $O(\lg^2 p)$. *More accurately, if* $p \geq 2$, *each packet has at most* $\lceil \kappa \lg^2 p \rceil + 1$ *items each containing one load value (representing an element count) and one key value, where* $\kappa = 1/(1 - \delta/2)$, $\delta = 1/(\lg e \lg p)$, *and* e *is the base of the natural logarithm, approximately* 2.718. *Note that if* $p \to \infty$ *then* $\delta \to 0$ *and* $\kappa \to 1$.

U3 *The total time is* $O(\log^3 p + (\log p)(\log N))$, *if we make the following two assumptions:*

- *It takes* $O(1)$ *time to send each piece of data.*
- *Each (leaf) processor maintains elements based on the priority queue described in Section 4.2.1 by letting the grain size of each element (corresponding to a task) be one.*

The PRS algorithm that satisfies these properties is very efficient for the following reasons. Concerning the first property, using only one combining operation and one disseminating operation is quite efficient because the number of sends/receives and the number of critical-path sends/receives are both optimal. This advantage is especially important on a network-based multicomputer, where each send/receive incurs significant overhead.

The second property shows that the packet size is moderate. Assume that each item requires 8 bytes. Then, for example, if p is $1,000$, the packet size is about 1 kilobyte; if p is $1,000,000$, the packet size is only about 4 kilobytes. In many networks, the time for sending a packet with 4 kilobytes is only a small number of times longer than that for a single word (see [27]). From the above two properties, we conclude that our PRS algorithm is very efficient in network-based multicomputers.

The third property shows that the time complexity for a parallel system with fast communication, like the CM5 [99], is also quite low.

In the remaining sections, we will design a PRS algorithm which satisfies the above three properties, U1-U3. This algorithm uses only one combining operation and one disseminating operation (property U1), in which each packet size satisfies property U2. The combining operation, described in Section 5.2.3, combines the key value distribution lists (defined in Section 5.2.2) of all the processors into the root processor. Note that the data of these lists roughly represent the key value distributions; these lists can be merged without too much loss of accuracy. In Section 5.2.3, we will also show that on the root processor we can select a key value threshold π_{thr} from the final combined list (which roughly represents the key value distribution of the entire system), such that the total number of elements with priorities $\leq \pi_{thr}$ is $\Theta(M)$. After the combining operation, we can simply disseminate π_{thr} to all the processors to select all elements with priorities $\leq \pi_{thr}$. Thus, this algorithm solves the PRS problem. In Section 5.2.4, we will also prove that the algorithm satisfies the third property U3. Finally, in Section 5.2.5, we will discuss the case in which the degree of processor tree is a constant (integer) more than two, and also describe how the algorithm can be applied to our multilist system, as described in Section 4.1.2.2.

5.2.2 Key Value Distribution Lists

In our PRS algorithm, each processor P_i needs to generate a list λ_i which can roughly represent the key value distribution in the whole processor subtree rooted at P_i. We will call such a list a *key value distribution list* or a *KVD list*. A KVD list has several items and each item contains a pair of values: a key value and a load value. For analyzing a KVD list, we will use the following notation:

- m_i: denotes the number of items in the KVD list λ_i.
- (π_{ij}, L_{ij}): denotes the jth item in the KVD list λ_i, where $1 \leq j \leq m_i$. Here, π_{ij} and L_{ij} are the key value and the load value of this item, respectively.
- Γ_i: denotes the processor subtree rooted at P_i.
- $n_i(\pi)$: denotes the number of elements with the key value π in the processor subtree Γ_i.
- $N_i(\pi)$: denotes $\sum_{\pi' \leq \pi} n_i(\pi')$.

For simplicity, if the root processor is P_{rt}, we will use $n(\pi)$ and $N(\pi)$ to stand for $n_{rt}(\pi)$ and $N_{rt}(\pi)$, respectively.

From the above definition of $N_i(\pi)$, we will have the following basic properties:

- $N_i(\infty)$ is the total number of elements in the processor subtree Γ_i while $N_i(-\infty)$ is zero, where ∞ ($-\infty$) is a value higher (lower) than any key value which can be used.
- $N_i(\pi_1) \leq N_i(\pi_2)$ if $\pi_1 \leq \pi_2$. That is, the function $N(\pi)$ is monotonically increasing.
- If processor P_i is an internal processor and processors P_l and P_r are the two children of P_i, then $N_i(\pi) = N_l(\pi) + N_r(\pi)$ for each π. This is due to the assumption that each internal processor contains no elements.

Definition 5.4 *From above, a KVD list λ_i (generated on processor P_i) is called a k-deviant KVD list, if the following two properties hold for the list:*

V1 *The key values are strictly increasing and the load values are monotonically increasing with the following restrictions.*

1. *For the final item* (π_{i,m_i}, L_{i,m_i}), *if* $\pi_{i,m_i} = \infty$, *then* $L_{i,m_i} = N_i(\infty)$; *otherwise,* $M \leq L_{i,m_i} \leq N_i(\pi_{i,m_i})$.
2. *All the non-final load values (i.e., all the load values except for the final one) are less than* M.

V2 *Let the list have a pseudo item* (π_{i0}, L_{i0}), *where* $\pi_{i0} = -\infty$, *and* $L_{i0} = 0$. *For each key* π, *where* $\pi_{ij} \leq \pi < \pi_{i(j+1)}$ *and* $0 \leq j \leq m_i - 1$, *the value* $N_i(\pi)$ *is in the range,* $L_{ij} \leq N_i(\pi) \leq kL_{ij}$.

From the above definition, if $k < k'$, a k-deviant KVD list is obviously also a k'-deviant list.

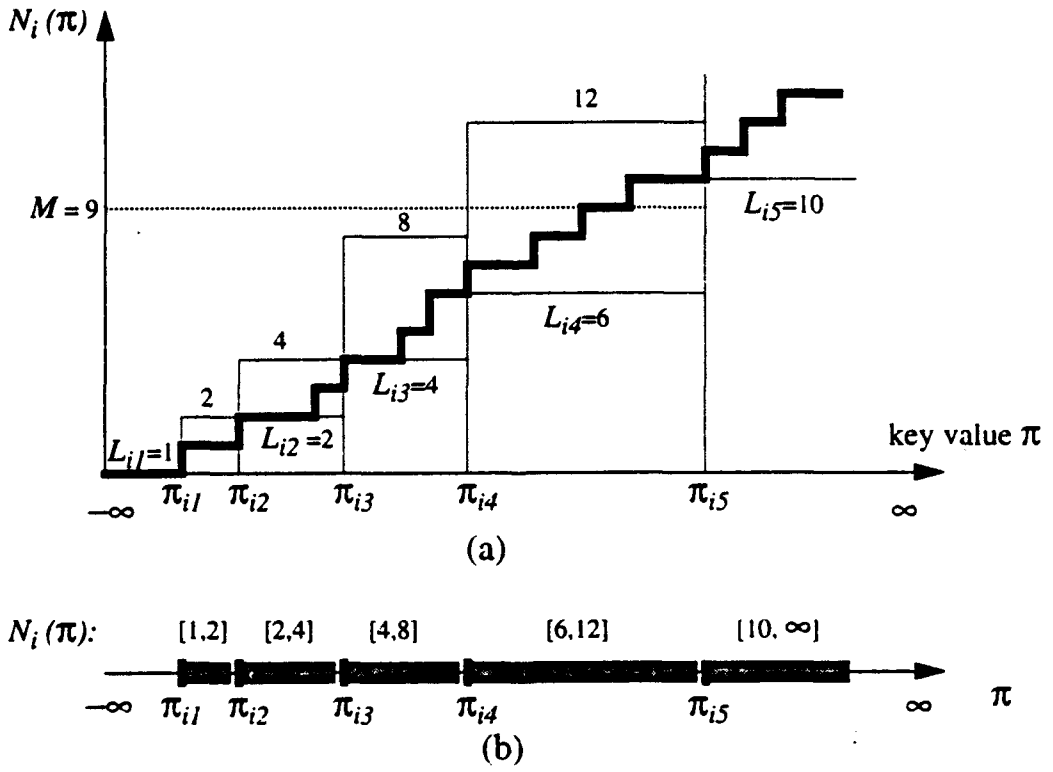


Figure 5.9: (a) A key value distribution in the processor subtree Γ_i, showing a KVD list λ_i (containing five items) which is 2-deviant, given $M = 9$. (b) Simplified diagram to show the possible range of $N_i(\pi)$, given the 2-deviant KVD list in (a).

Figure 5.9(a) illustrates a key value distribution in the processor subtree Γ_i, showing a KVD list λ_i (containing five items) which is 2-deviant, given $M = 9$.

An important feature of a k-deviant KVD list λ_i is that for each key $\pi < \pi_{i,m_i}$ (the final key value) we can find a load value L from the KVD list, such that $L \leq N_i(\pi) \leq kL$. In Figure 5.9(a), the shadowed area indicates the possible range of $N_i(\pi)$ for each π. (Note that given a 2-deviant list the possible range of each $N_i(\pi)$ can also be depicted in a simpler way in Figure 5.9(b).) Thus, the list provides us with the rough KVD information.

In the next section, we will create KVD lists containing an exponentially increasing series of load values. Thus, the number of elements in a KVD list can be reduced to a very small number.

5.2.3 Combining

In this section, we will first design a combining operation which can find a key value threshold π_{thr} satisfying the condition $N(\pi_{thr}) = \Theta(M)$, and in which each packet has at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items (where κ is defined in Theorem 5.6). Then, we will further improve the operation such that each packet only needs at most $\lceil \kappa \lg^2 p \rceil + 1$ items (note that M may be much larger than p), while we can still find a key value threshold π_{thr} satisfying $N(\pi_{thr}) = \Theta(M)$.

We summarize the combining operation in which each packet has at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items, as follows. Each leaf processor first creates a 2-deviant KVD list with at most $\lceil \lg M \rceil + 1$ items, as described in Section 5.2.3.1, and then sends the list to its parent. Then, for each internal processor node P_i, if its two children have generated k-deviant KVD lists and have sent their lists to P_i, then P_i can, as shown in Section 5.2.3.2, merge their lists into a KVD list with $k(1+\delta)$-deviation and with at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items. If processor P_i is not the root, the list will be sent to its parent which in turn repeats the above operation. If processor P_i is the root P_{rt}, Lemma 5.7 below proves that the KVD list λ_{rt} of the root processor is 4-deviant. Now, we can choose the final key value $\pi_{rt,m_{rt}}$ in λ_{rt} as the key value threshold because the condition $N(\pi_{rt,m_{rt}}) = \Theta(M)$ holds according to Lemma 5.8.

Lemma 5.7 *For the combining operation described above, the KVD list on the root processor is 4-deviant.*

Proof. By Lemma 5.9 in Section 5.2.3.1, all the KVD lists on leaf processors (at level q) are 2-deviant. Then, the KVD lists on those internal processor nodes at level $q-1$ are $2(1+\delta)$-deviant, by Lemma 5.11 in Section 5.2.3.2. Whenever we go up one level, the deviation degree becomes $(1+\delta)$ times larger. Thus, the list on the root processor will become $2(1+\delta)^q$-deviant. Since[3] $2(1+\delta)^q \leq 4$, the list is also a 4-deviant list. □

Lemma 5.8 *For the combining operation described above, $N(\pi_{rt,m_{rt}}) = \Theta(M)$, where $\pi_{rt,m_{rt}}$ is the final key value in the KVD list λ_{rt} of the root processor P_{rt}.*

Proof. We will prove that (1) $N(\pi_{rt,m_{rt}}) \geq M$ and (2) $N(\pi_{rt,m_{rt}}) \leq 4M$.

(1) We will prove $N(\pi_{rt,m_{rt}}) \geq M$ from the first restriction of Property V1 in the KVD list λ_{rt} (note that the list is 4-deviant from Lemma 5.7). If $\pi_{rt,m_{rt}} = \infty$, then $N_{rt}(\infty) \geq M$ because the total number of elements, $N(\infty)$, is at least M from the definition of the PRS problem. If $\pi_{rt,m_{rt}}$ is not ∞, $M \leq N_{rt}(\pi_{rt,m_{rt}})$. Thus, $N(\pi_{rt,m_{rt}}) \geq M$, for both cases.

(2) Since $N(\pi_{rt,m_{rt}}) = N(\pi_{rt,m_{rt}} - 1) + n(\pi_{rt,m_{rt}})$, it suffices to prove that the following two conditions hold: (a) $n(\pi_{rt,m_{rt}}) \leq 1$ and (b) $N(\pi_{rt,m_{rt}} - 1) < 4M$. Since all elements have distinct key values, each $n(\pi) \leq 1$, i.e., the condition (a) holds. Since the KVD list λ_{rt} is 4-deviant, $N(\pi_{rt,m_{rt}} - 1) \leq 4L_{rt,m_{rt}-1}$. From the second restriction of property V1, $L_{rt,m_{rt}-1} < M$. Thus, $N(\pi_{rt,m_{rt}} - 1) < 4M$, i.e., condition (b) holds. □

The above combining operation can find a key value threshold satisfying the condition $N(\pi_{thr}) = \Theta(M)$, with each packet requiring at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items. This upper

[3]The Maclaurin series (Taylor expansion around zero) for $\log(1+\delta)$ is $\delta - \frac{\delta^2}{2} + \frac{\delta^3}{3} - \frac{\delta^4}{4} \ldots$. In addition, since $p \geq 2$, $1 > \delta > 0$. Hence, we can derive that $\delta > \log(1+\delta) > \delta - \frac{\delta^2}{2}$. Thus, $(1+\delta)^q = 2^{q \lg(1+\delta)} < 2^{q\delta(\lg e)} = 2$.

bound is greater than $\lceil \kappa \lg^2 p \rceil + 1$ (in Property U2) when $M > p$. So, we will, in Section 5.2.3.3, reduce the number of items in each list, such that each packet has at most $\lceil \kappa \lg^2 p \rceil + 1$ items (satisfying Property U2) while retaining the property that the key value $\pi_{rt,m_{rt}}$ is a key value threshold satisfying the PRS problem.

5.2.3.1 Leaf Processors

In this section, we will design the algorithm, called the Create Algorithm (below), that each leaf processor will use to create its KVD list. Then, we will prove in Lemma 5.9 that the created KVD list is 2-deviant, and in Lemma 5.10 that the list has at most $\lceil \lg M \rceil + 1$ items.

Create Algorithm

Step 1 Initially, let the variable $L = 1$ and the list λ_i be empty.

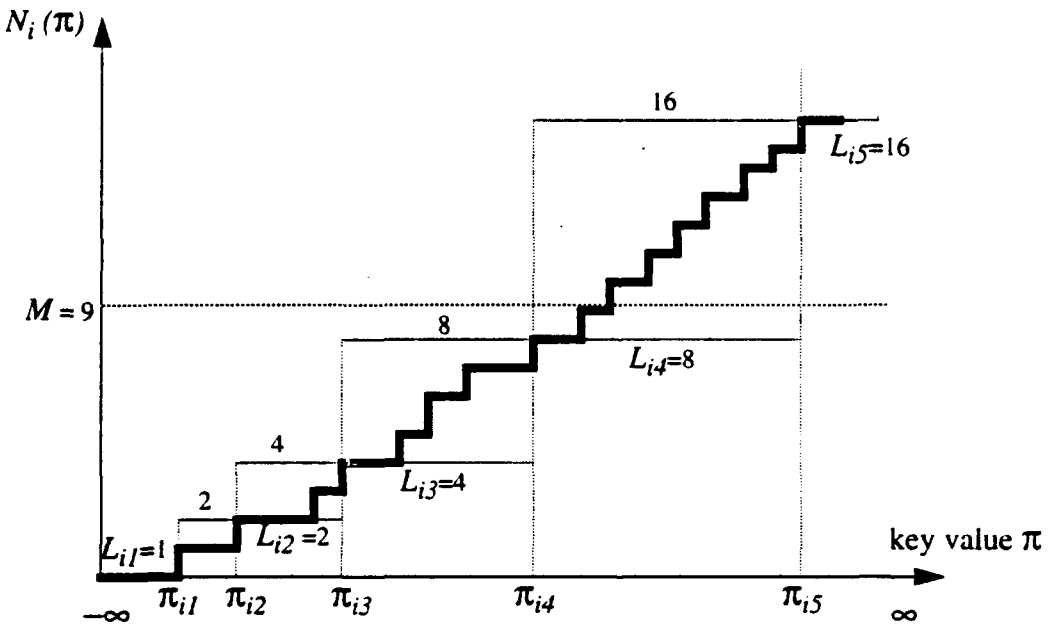


Figure 5.10: A key value distribution in the processor subtree Γ_i, showing the 2-deviant KVD list λ_i (containing five items) which is generated by the Create Algorithm, given $M = 9$.

Step 2 If $N_i(\infty) < L$, append a new item $(\infty, N_i(\infty))$ to the end of the list λ_i and then stop.

Step 3 Find the Lth smallest key value π, and append a new item (π, L) to the end of the list λ_i. Note that since all elements have distinct key, $N_i(\pi) = L$.

Step 4 If $L \geq M$, stop; otherwise, let $L = 2L$ and repeat Step 2.

From the above algorithm, all the non-final items in this list should be generated at Step 3 and hence their load values are 1, 2, 4, ..., and $2^{m_i-2}(< M)$. Figure 5.10 illustrates a key value distribution in the processor subtree Γ_i rooted at processor P_i, showing the 2-deviant KVD list λ_i (containing five items) which is generated by the Create Algorithm, given $M = 9$.

Lemma 5.9 *On each leaf processor, its KVD list generated by the Create Algorithm is 2-deviant.*

Proof. We will prove that on each leaf processor P_i, its KVD list λ_i generated by the Create Algorithm satisfies Properties V1 and V2.

V1 We will consider the non-final items first and then the final item. From the Create Algorithm, all the non-final load values are 1, 2, ..., $2^{m_i-2}(< M)$, so they are strictly increasing and the second restriction holds. As for all non-final key values π_{ij}, since $L(\pi_{ij}) = L_{ij}$ and non-final load values are strictly increasing, we can derive that these non-final key values are also strictly increasing.

Now, let us consider the final item (π_{i,m_i}, L_{i,m_i}). If the algorithm stops at Step 2, then $\pi_{i,m_i} = \infty$ and $L_{i,m_i} = N_i(\infty)$. Otherwise, the algorithm stops at Step 3. In this case, $\pi_{i,m_i} \neq \infty$ and $L_{i,m_i} = N_i(\pi_{i,m_i}) \geq M$. Thus, the first restriction holds. In addition, since the last load value is either $N_i(\infty)$ (greater than or equal to each $N_i(\pi)$ and each non-final load value, which is some $N_i(\pi)$) or at least M, the value is no less than any non-final load value. Thus, all the load values in the list are monotonically increasing. Since the last key value is either ∞ or a key value π such that $N_i(\pi) \geq M$, we can derive that the key value is greater than any non-final key value. Thus, all the key values in the list are strictly increasing.

V2 Consider any two consecutive items, the jth item and the $(j+1)$-st item, where $0 \leq j \leq m_i - 1$. Since $L = N_i(\pi)$ for each item (π, L) generated by the Create Algorithm, we can derive the following property: for each key value π, where $\pi_{ij} \leq \pi < \pi_{i(j+1)}$, $L_{ij} = N_i(\pi_{ij}) \leq N_i(\pi) \leq N_i(\pi_{i(j+1)}) = L_{i(j+1)}$. In order to prove that Property V2 holds, we will only prove that $L_{i(j+1)} \leq 2L_{ij}$, as follows. If the $(j+1)$-st item is generated at Step 2, then $L_{i(j+1)} = N_i(\infty) < 2L_{ij}$; otherwise, the item is generated at Step 3 and $L_{i(j+1)} = 2L_{ij}$. □

Lemma 5.10 *On each leaf processor, its KVD list obtained by the Create Algorithm has at most* $\lceil \lg M \rceil + 1$ *items.*

Proof. Since the non-final values are 1, 2, ..., 2^{m_i-2} ($< M$), we can obtain that $m_i - 2 < \lg M$, i.e., $m_i \leq \lceil \lg M \rceil + 1$. Thus, the lemma holds. □

5.2.3.2 Internal Processor Nodes

In this section, we will design an algorithm that each internal processor node P_i uses to merge the KVD lists from both of its children P_l and P_r into its KVD list λ_i. We will also prove that the new list is $k(1+\delta)$-deviant if both KVD lists from its children are k-deviant, and that the new list has at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items. But, before investigating the algorithm, we will first present a simpler merge algorithm as follows.

Simple Merge Algorithm: Let the symbols (π_l, L_l) and (π_r, L_r) represent the first items in λ_l and λ_r respectively; and let the symbols (π'_l, L'_l) and (π'_r, L'_r) represent the previous deleted items in λ_l and λ_r respectively.

Step 1 Let both (π'_l, L'_l) and (π'_r, L'_r) be $(-\infty, N_i(-\infty) = 0)$.

Step 2 If $\pi_l = \pi_r$, append a new item $(\pi_l, L_l + L_r)$ to the end of λ_i, remove both first items from λ_l and λ_r, and go to Step 4. Note that if a new item is appended then the values of π_l, L_l, π_r, L_r, π'_l, L'_l, π'_r, and L'_r, are changed implicitly.

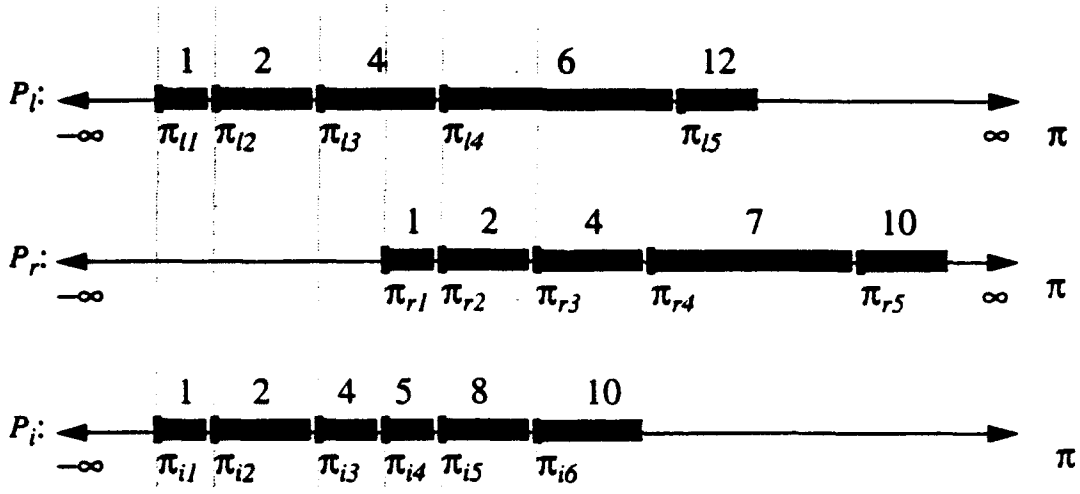


Figure 5.11: An example of the merge operation from λ_l and λ_r into λ_i, given $M = 9$.

Step 3 If $\pi_l < \pi_r$, append a new item (π_l, L_l+L'_r) to the end of λ_i and remove the first item of λ_l; otherwise, append a new item (π_r, L_r+L'_l) to the end of λ_i and remove the first item of λ_r. Note that after this step the values of π_l, L_l, π_r, L_r, π'_l, L'_l, π'_r, and L'_r, are changed implicitly.

Step 4 In the newly appended item (π, L), if $L \geq M$ or $\pi = \infty$, stop; otherwise, repeat Step 2.

Figure 5.11 illustrates an example of the merge operation. We also need to point out that this merge algorithm will not repeat Step 2 (from Step 4) in the condition that either of KVD list λ_l or λ_r is empty. Assume that the final key value of one list, say λ_r, is merged into λ_i. From the first restriction of property V1, either the final key value of λ_r is ∞ or the final load value of λ_r is at least M. For the former case, the appended key value of λ_i is also ∞ and therefore the appended item is the final item (see Step 4) in the new list; for the latter case, the appended load value is also at least M and therefore the appended item is also the final item (also see Step 4). From the above, the last key value of λ_i must be no greater than either of the last key values of λ_l and λ_r.

Lemma 5.11 (below) proves that if the deviation degrees of the lists from its children are the same, the parent's list has the same deviation degree. Since the lists of all leaf processors are all 2-deviant, the list on the root will still be 2-deviant.

Lemma 5.11 *For the above simple merge algorithm, if both λ_l and λ_r are k-deviant, the new list λ_i is also k-deviant.*

Proof. We will examine Properties V1 and V2 in the new list λ_i.

V1 The above simple merge algorithm merges items of both lists λ_l and λ_r according to their key values. Step 2 merges two items with the same key value into one item, while Step 3 keeps the key values of the new list λ_i in increasing order. Thus, in the new list, the key values are strictly increasing. As for the load values in the list λ_i, they are monotonically increasing for the following reason. For each item in λ_i, its load value must be any of L_l+L_r (Step 2), L'_l+L_r, and L_l+L'_r (Step 3), while the load value of its previous item (if any) is $L'_l + L'_r$. Since the load values in both λ_l and λ_r are monotonically increasing, $L_l \geq L'_l$ and $L_r \geq L'_r$. Hence, the load value of an item must be no less than that of its previous item. Therefore, the load values in λ_i are monotonically increasing too. In addition, we will prove that the two restrictions in property V1 hold, as follows.

1. For the final item, (π_{i,m_i}, L_{i,m_i}), suppose that $\pi_{i,m_i} = \infty$. The final items of λ_l and λ_r also have key values ∞. Hence, both final load values of λ_l and λ_r are $N_l(\infty)$ and $N_r(\infty)$. Thus, from Step 2, the final load value of λ_i is $N_l(\infty) + N_r(\infty) = N_i(\infty)$, i.e., the first restriction holds in this case.

   Suppose that $\pi_{i,m_i} \neq \infty$. Then, from Step 4, the final load value L_{i,m_i} must be at least M. So, for the first restriction, we will only need to prove that the following condition C holds: $L_{i,m_i} \leq N_i(\pi_{i,m_i})$. If π_{i,m_i} is less than both π_{l,m_l} and π_{r,m_r}, we can derive the condition C from Property V2 (which will be shown below). If π_{l,m_l} or π_{r,m_r}, say the former, is π_{i,m_i} (note that π_{i,m_i} cannot be greater than either π_{l,m_l} or π_{r,m_r} as shown earlier), then $L_{l,m_l} \leq N_l(\pi_{l,m_l})$ from the first restriction of Property V1 in the list λ_l. Hence, we can derive that the condition C holds, by applying the technique used in the proof for Property V2 (below). Therefore, this restriction holds.

2. All the non-final load values in the list λ_i are less than M because if one of them were at least M the item containing it would be the final item (see Step 4).

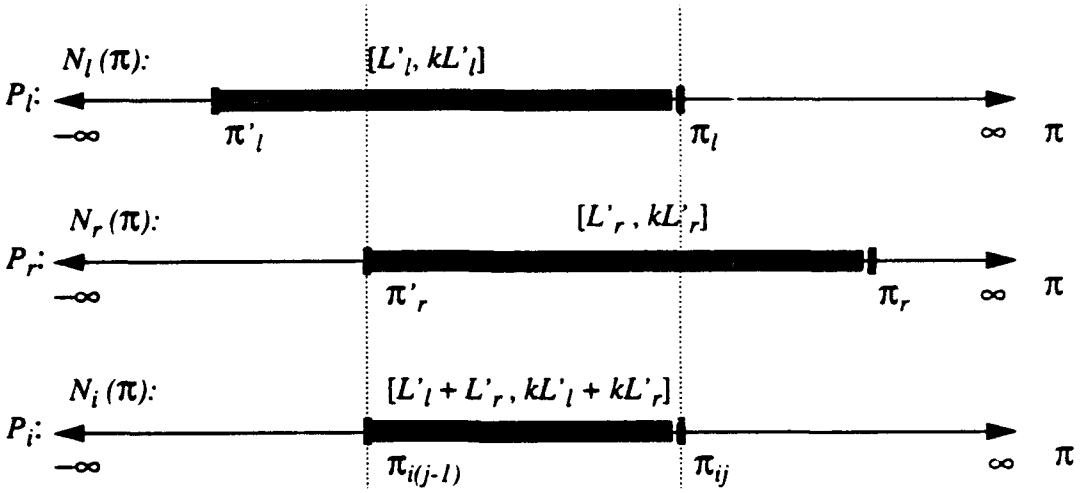


Figure 5.12: An example of examining property V2.

V2 In the above merge algorithm, consider the moment immediately before we append the jth item to λ_i, where $1 \leq j \leq m_i$. The key value of the $(j-1)$-st item is $\pi_{i(j-1)} = \max(\pi'_l, \pi'_r)$ and the key value of the jth item will be $\pi_{ij} = \min(\pi_l, \pi_r)$. Figure 5.12 illustrates an example. Thus, for all key π, where $\pi_{i(j-1)} \leq \pi < \pi_{ij}$, the condition $L'_l \leq N_l(\pi) \leq kL'_l$ holds because λ_l is k-deviant; the condition $L'_r \leq N_r(\pi) \leq kL'_r$ holds because λ_r is k-deviant. Since $N_i(\pi) = N_l(\pi) + N_r(\pi)$, the condition $L'_l + L'_r \leq N_i(\pi) \leq kL'_l + kL'_r$ holds. Since the $(j-1)$-st load value is $L'_l + L'_r$, property V2 holds. □

Although this algorithm can merge KVD lists while keeping the same deviation degree, the number of items in the new list may double after each merge operation. Hence, the list of the root processor and those internal processor nodes near the root may have $O(p \lg M)$ items. Thus, the performance will degrade seriously, given a large p.

In order to solve this problem, we will modify the above simple merge algorithm by removing those items (excluding the first and final items) whose load values are too "close" to the previous load values. More precisely, at Step 4, we will add an operation immediately

before "repeat Step 2" as follows: delete the newly appended item (π, L) if $L < (1+\delta)L'$, where L' is the load value of the previous item. Lemma 5.13 (below) will prove that the number of items can be greatly reduced to at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$. The price that we have to pay for reducing the number of items is that the list will have a higher deviation degree. Lemma 5.12 proves that the deviation degree becomes only $(1+\delta)$ times higher. Since δ is a very small number ($\delta = 1/(2 \lg e \lg p)$), we can still keep the deviation degree of the KVD list λ_{rt} on the root processor constant (as shown in Lemma 5.7).

Lemma 5.12 *In the above modified merge algorithm, if both λ_l and λ_r are k-deviant, the new list λ_i is $k(1+\delta)$-deviant.*

Proof. We will prove that for the new list λ_i Properties V1 and V2 hold.

V1 For the modified merge algorithm, we still keep the first and final items, and may delete some items between them. Since the ordering of the remaining items is still not changed and the final item is not deleted, we can derive that the new KVD list still satisfies Property V1.

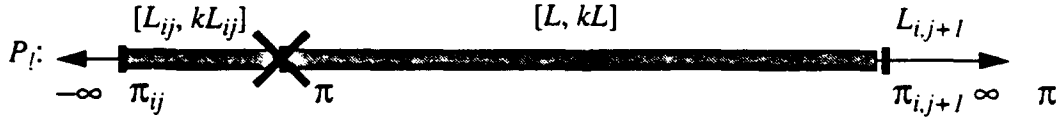


Figure 5.13: An example of removing items.

V2 In the KVD list λ_i, consider any two consecutive items, say the jth item and the $(j+1)$-st item. Let the item (π, L) be the last removed item between the jth and the $(j+1)$st items, as shown in Figure 5.13. In the modified merge algorithm, we remove the item only if $L < (1+\delta)L_{ij}$. Thus, for each key value π, $\pi_{ij} \le \pi < \pi_{i(j+1)}$, $L_{ij} \le N_i(\pi) \le kL < k(1+\delta)L_{ij}$. Thus, Property V2 holds. □

Lemma 5.13 *For an internal processor node P_i, if its KVD list λ_i is merged by using the above modified merge algorithm, the number of items in λ_i is at most* $\lceil \kappa(\lg p)(\lg M) \rceil + 1$.

Proof. For this proof, it suffices to prove that the number of non-final items is $n \leq \lceil \kappa(\lg p)(\lg M) \rceil$. For every two consecutive non-final load values L' and L (the next load value of L') in λ_i, the condition $(1+\delta)L' \leq L$ holds from the modified merge algorithm. Hence, the last one among all the non-final load values must be at least $(1+\delta)^{n-1}$ (note that the first load value must be at least one). Since each non-final load value is less than M from the second restriction of Property V1, we can derive that $(1+\delta)^{n-1} < M$ or $n - 1 < (\lg M)/\lg(1+\delta)$. From this result, it can be verified[4] that $n \leq \lceil \kappa(\lg p)(\lg M) \rceil$. □

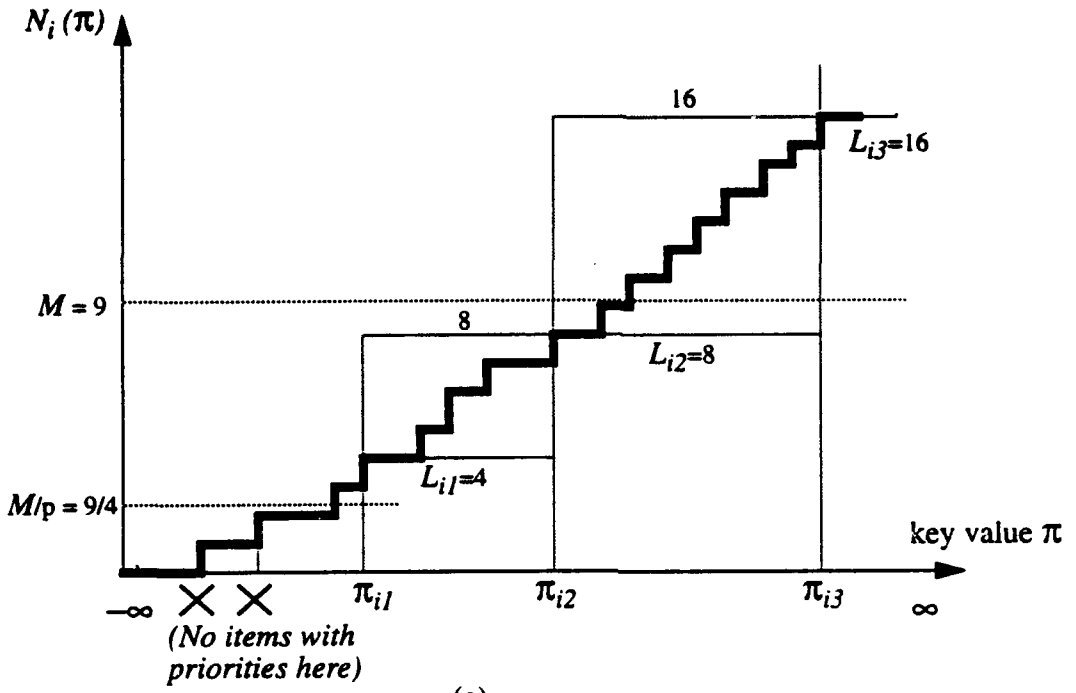
5.2.3.3 Improved Combining Operation

For the above combining operation described in the previous two sections, each packet has at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$ items. When $M > p$, the upper bound of the item number is higher than $\lceil \kappa \lg^2 p \rceil + 1$ in Property U2. So, in this section, we want to reduce the upper bound to $\lceil \kappa \lg^2 p \rceil + 1$, while letting the final key value in the list of the root processor be a key value threshold.

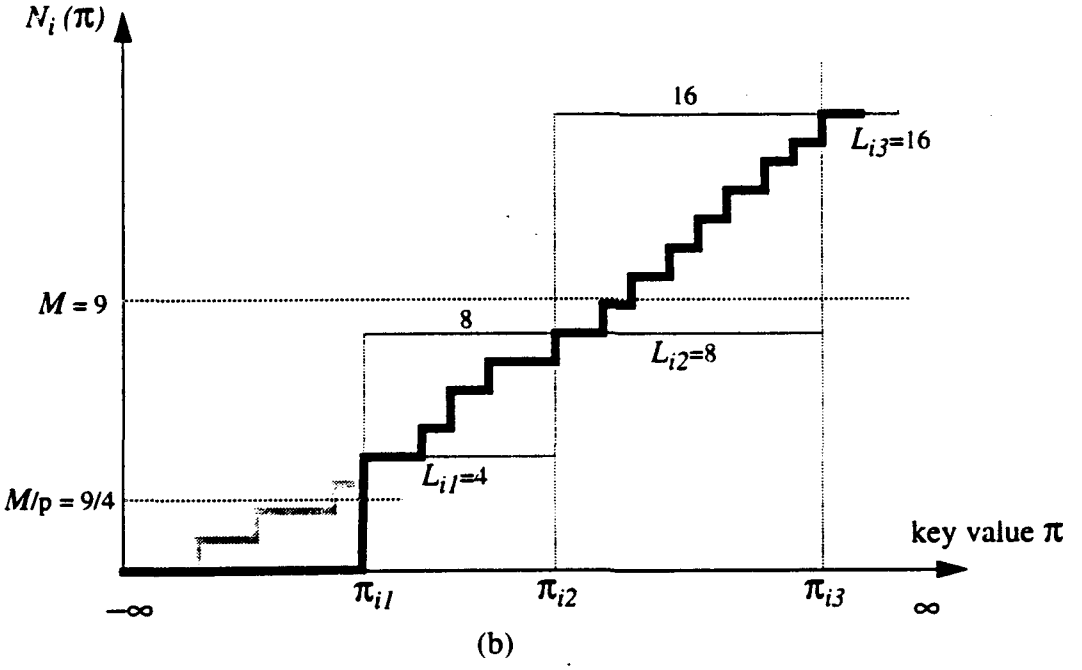
In the improved combining operation, we only need to modify the original Create Algorithm by removing those non-final items whose load values are less than M/p, as illustrated in Figure 5.14(a) with $M = 9$ and $p = 4$. Hence, the first non-final item (if any) has a load value L, $M/p \leq L < 2M/p$. Thus, the number of non-final items generated by the new Create Algorithm will become at most $\lceil \lg p \rceil$ (i.e., the total number of items are at most $\lceil \lg p \rceil + 1$), because we can double at most $\lceil \lg p \rceil - 1$ times between M/p (inclusive) and M (exclusive).

On the internal processor nodes, we still use the modified merge algorithm (described in the previous section) to merge the KVD lists. It can be easily verified from the modified merge algorithm and the new Create Algorithm that the first non-final load value (if any) of a KVD

[4] As shown earlier, $\log(1+\delta) > \delta - \delta^2/2 = \delta(1 - \delta/2) = \delta/\kappa$. Hence, $(\lg M)/\lg(1+\delta) < (\lg M)/((\lg e)(\delta/\kappa)) < \kappa(\lg p)(\lg M)$.



(a)



(b)

Figure 5.14: (a) Removing those items with priorities lower than $M/p(= 9/4)$. (b) Increasing the key values less than π_{i1} to π_{i1}.

list on an internal processor node is still at least M/p. Then, we can derive in Lemma 5.14 that the number of items in λ_i is at most $\lceil \kappa \lg^2 p \rceil + 1$.

Lemma 5.14 *For the above improved combining operation, if a KVD list λ_i on an internal processor P_i is generated by the modified merge algorithm, the number of items in λ_i is at most $\lceil \kappa \lg^2 p \rceil + 1$.*

Proof. Since the first non-final node (if any) is at least M/p and each non-final node is less than M, we can derive that the number of items in λ_i is at most $\lceil \kappa(\lg p)(\lg M) \rceil + 1$, by applying the same technique used in the proof of Lemma 5.13. □

Although the number of items in each packet is reduced to what we want, we should note that after the modification each KVD list generated by the new Create Algorithm may not be 2-deviant. This is because on the corresponding leaf processor there may be some elements with key values lower than the first key value of the list. In order to let each leaf processor's KVD list λ_i become 2-deviant again, we increase those key values less than π_{i1} (in the processor subtree Γ_i) to π_{i1}, as illustrated in Figure 5.14(b). After increasing the key values, each $n(\pi)$ becomes at most $2M/p$.

Since the internal processor nodes still use the modified merge algorithm to generate their KVD lists, Lemma 5.11 still holds and then the KVD list in the root processor is still 4-deviant from Lemma 5.7. Although $n(\pi) \leq 2M/p$ for each π, we still can derive that $N(\pi_{rt,m_{rt}}) = \Theta(M)$, by applying the same technique used in the proof of Lemma 5.8.

The above result only shows that *after* increasing key values, the value $N(\pi_{rt,m_{rt}})$ is $\Theta(M)$. Lemma 5.15 proves that if we increase at most $2M/p$ nodes on each of p leaf processors and the value $N(\pi)$ is $\Theta(M)$ after increasing these key values, the value $N(\pi)$ is also $\Theta(M)$ before the increasing. This implies that the condition $N(\pi_{rt,m_{rt}}) = \Theta(M)$ also holds before the increasing; that is, the final key value $\pi_{rt,m_{rt}}$ is a key value threshold satisfying the PRS problem.

Lemma 5.15 *For the PRS problem, suppose that we increase key values of at most $2M/p$ elements on each of the p leaf processors. If $N(\pi) = \Theta(M)$ after we increase these key values, then the original $N(\pi)$ (before increasing these key values) is also $\Theta(M)$.*

Proof. Let $N_{before}(\pi)$ denote the original value $N(\pi)$ and $N_{after}(\pi)$ denote the value $N(\pi)$ after we increase some key values. Since we increase key values of at most $p(2M/p)$ elements, the value $N(\pi)$ decreases at most $2M$ in total. Hence, $N_{before}(\pi) - 2M \leq N_{after}(\pi) \leq N_{before}(\pi)$, that is, $N_{after}(\pi) \leq N_{before}(\pi) \leq N_{after}(\pi) + 2M$. Therefore, if $N_{after}(\pi) = \Theta(M)$, then $N_{before}(\pi) = \Theta(M)$. □

5.2.4 Time Complexity

In this section, we will analyze the time complexity of the PRS algorithm based on the two assumptions: (1) it takes $O(1)$ time to send each piece of data; (2) each (leaf) processor maintains elements based on the priority queue described in Section 4.2.1 by letting the grain size of each element (corresponding to a task) be one.

The KVD lists on leaf processors are generated by the Create Algorithm. In this algorithm, Step 3 can use the THRESHPRI operation (an operation on the priority queue described in Section 4.2.1), so the computation time at Step 3 is $O(\log N_i)$, where N_i is the total number of elements on processor P_i. Note that each other step only takes $O(1)$ time. Since $N_i \leq N$, the time complexity is $O(\log N)$ too. Since there are at most $\lceil \lg p \rceil + 1$ items in the list on a leaf processor, we will repeat each step at most $\lceil \lg p \rceil + 1$ times. So, the total time on each leaf processor is $O((\log p)(\log N))$.

The KVD lists on internal processors are generated by the modified merge algorithm. Since the algorithm does a merge operation, the time depends on the number of items. The number of items in each list is at most $\lceil \lg^2 p \rceil + 1$ (or $O(\log^2 p)$), so the total time taken for each internal processor is only $O(\log^2 p)$. Since there are $\lg p$ levels in the processor tree and processors at the same level can process the merge operation in parallel, the total time of the combining operation is $O(\log^3 p + (\log p)(\log N))$.

As for the disseminating operation, we broadcast the value of π_{thr} from the root processor

to each leaf processor. Obviously, the time complexity is smaller than that of the combining operation.

After each leaf processor receives the value π_{thr}, the processor will use the SPLIT operation (an operation of the priority queue described in Section 4.2.1) to partition the priority queue into two parts, one containing tasks with priorities $\pi \geq \pi_{thr}$ and the other containing the remaining tasks. The time complexity for the SPLIT operation is only $O(\log n)$.

From the above, we conclude that the total time complexity is $O(\log^3 p + (\log p)(\log N))$.

5.2.5 Discussion

In this section, we will first show that when the degree of the processor tree is a constant (integer) more than two, we still can solve the PRS problem by using one combining and disseminating operation in which each packet size is $O(\log^2 p)$. Then, we will discuss how to apply the PRS algorithm to our multilist scheduling system.

Suppose that an internal processor node P_i has three children. Processor P_i will do the 3-way merge operation after receiving three KVD lists from its children. Using the technique of the modified merge algorithm, the number of items in the KVD list λ_i is still $O(\log^2 p)$ and the deviation degree of λ still increases by a factor of $(1+\delta)$. Since the height of the processor tree is lower than $\lg p$, the KVD list in the root processor is still 4-deviant. Thus, the final key value of this list is still a key value threshold π_{thr} satisfying $N_i(\pi_{thr}) = \Theta(M)$. This can be generalized to any constant degree processor tree, so we still can solve the PRS problem by using packets with size $O(\log^2 p)$.

For the multilist scheduling model, the problem in Section 4.1.2.2 is as follows. Given a load threshold L_{thr} and a set of tasks each containing a priority and a grain size (represented by an integer), select a set of highest-priority tasks whose total load (summation of grain sizes) is $L_{sel} = \Theta(L_{thr})$, if each task's grain size is $O(L_{thr})$ and the total load of tasks is $L_{total} = \Omega(L_{thr})$. If the maximum task grain size G_{max} is larger than L_{thr}, then $L_{sel} = O(G_{max})$ because we may need to group the task with G_{max} (e.g., when the task with the grain size G_{max} has the highest priority). If the total load of tasks is $L_{total} < L_{thr}$, then $L_{sel} = L_{total}$ because we can at most group tasks with total load L_{total}.

Now, we want to translate the above problem to the PRS problem in order to apply the PRS algorithm presented in this section. Since each task $\mathcal{T}$ may have a different grain size $G_{\mathcal{T}}$, we can conceptually break a task into $G_{\mathcal{T}}$ elements with the same priority. Then, for each element which is the j-th element with priority π on processor P_i, we can define its key value as the compound key $(-\pi, i, j)$, such that all key values are distinct. Note that by negating π, we translate highest priority to lowest key value. We should note that when we break a task $\mathcal{T}$ into $G_{\mathcal{T}}$ elements, we let these elements be the ith element to the $(i + G_{\mathcal{T}} - 1)$-st element with the same priority of $\mathcal{T}$ on the same processor, so that the elements for the same task will have consecutive key values. Thus, there is at most one task whose elements are partially selected. Since in the actual implementation we still need to select the whole task $\mathcal{T}$, the total load of selected tasks may be G_{max} higher than the total number of selected elements. If $G_{max} \leq L_{thr}$, the total load of selected tasks is still $\Theta(L_{thr})$.

In addition, since the system does not know L_{total} in advance, we cannot guarantee that $L_{total} \geq L_{thr}$. In the case that $L_{total} < L_{thr}$, the final item of the KVD list in the root of the processor tree must be $(\infty, N(\infty) = L_{total})$ from the first restriction of property V1. In this case, we can simply broadcast the final key value (∞) to select all the tasks such that $L_{sel} = L_{total}$.

Chapter 6

Experimental Results

In this chapter, we will describe our experiments with the multilist scheduling model and show good performance results for the *PBFS-GPQ* and *PDC-WK* scheduling algorithms. (Since for parallel best-first search (BFS) we only choose *PBFS-GPQ* and for parallel divide-and-conquer (D&C) we only choose *PDC-WK*, we will respectively abbreviate their names as *PBFS* and *PDC* in this chapter for simplicity.) In Section 6.1, we will describe the environment in which we have run our experiments. Then, we will present two application examples, the Fibonacci problem and the set covering problem, each of which can use either the PBFS or the PDC scheduling algorithm, and we will report our experimental results for these applications. The two application examples will be described in Sections 6.2 and 6.3, respectively.

6.1 Environment

Our multilist scheduling model is currently implemented on Nectar [6], a high-bandwidth and low-latency computer network developed at Carnegie Mellon University. The Nectar system consists of a *Nectar network* and a set of network coprocessors, called *communication accelerator boards* (*CABs*), as illustrated in Figure 6.1. We can connect hosts (e.g., workstations) to Nectar by attaching them to CABs via VME buses. Each CAB is a Sparc-based network coprocessor with 1.5 Mbytes of local memory. Since each CAB has local memory, we can install some processes on a CAB. The Nectar network consists of 100 Mbits/s fiber-optic links,

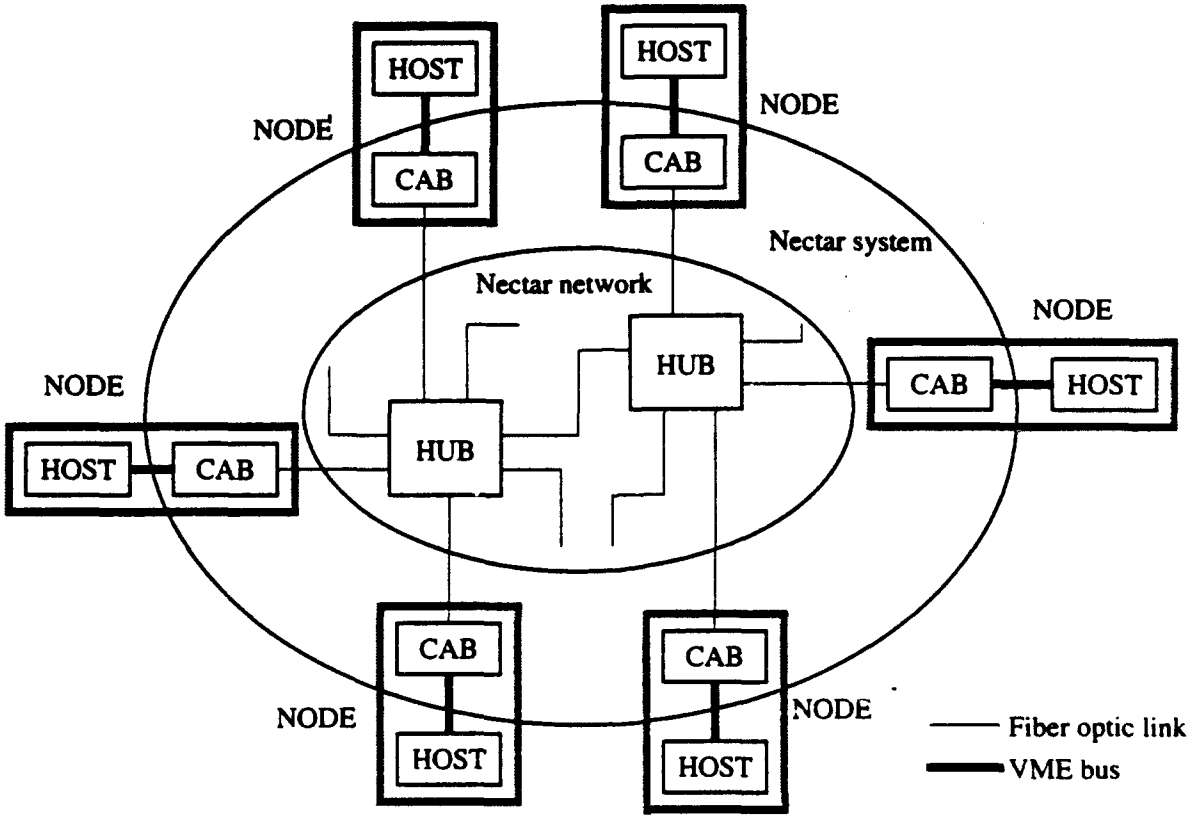


Figure 6.1: The Nectar system.

plus 16 × 16 crossbar switches called HUBs. The CAB is connected to a HUB via a fiber link. In our experiments, we use up to 8 Sun4/330s. The number of processors is not large, but is sufficient to validate our multilist scheduling approach. In order to make the results directly comparable, we run our experiments when no other users are using these machines. As for software, *Nectarine* [94] is an interface package that provides efficient communication primitives for programmers to access the Nectar runtime system. We have implemented our system using Nectarine.

Basically, the implementation of the multilist scheduling model follows the description presented in Chapter 4. For maintaining physical lists (PLs), we choose 2-4 trees instead of 2-3 trees simply because 2-4 trees may require slightly fewer operations of reconfiguring the trees. Note that for all integer constants $a \geq 2$ and $b > a$ the heights of all a-b trees are still $O(\log n)$ [3], where n is the number of distinct priority keys; thus, properties P1 and P2 in Section 4.2 still hold for all such a-b trees. In addition, since derived PLs have not been implemented, we

still use two base PLs to implement the *PDC* scheduling algorithm.

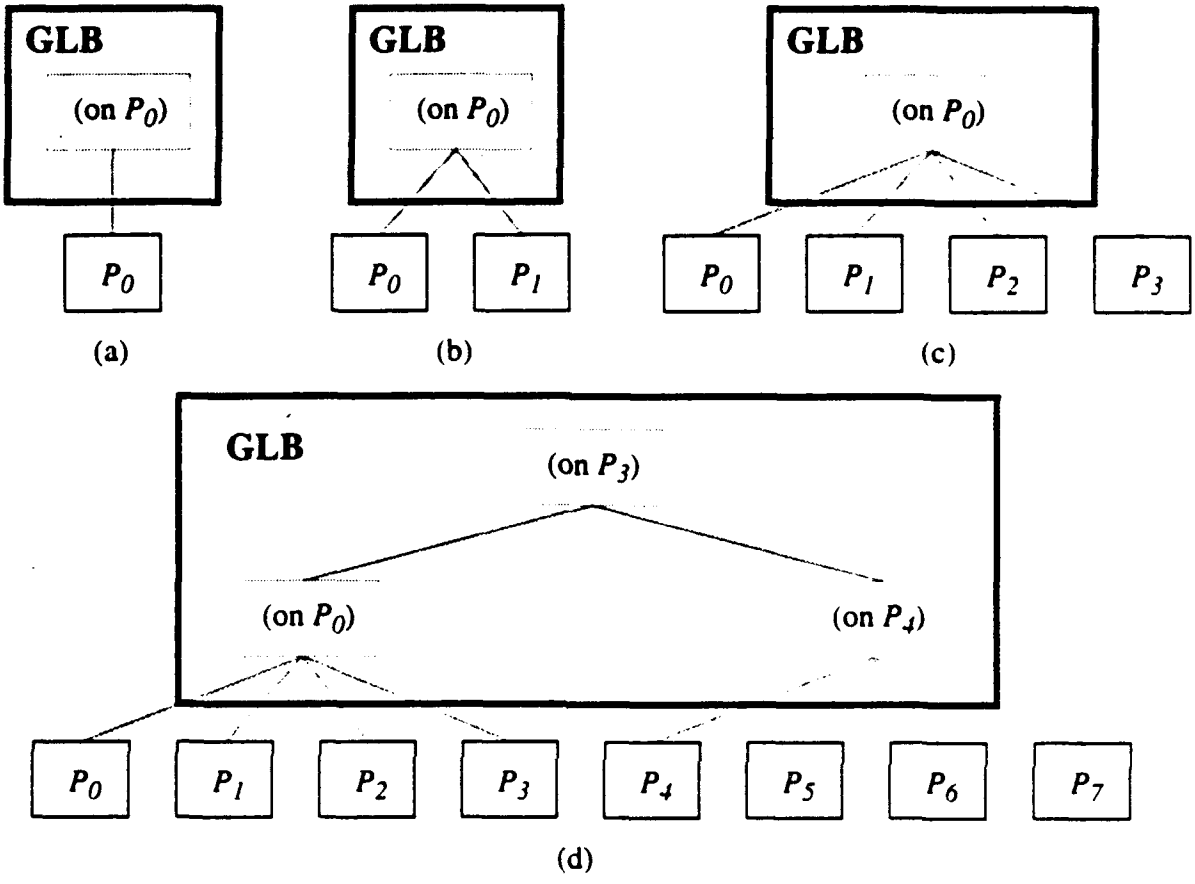


Figure 6.2: GLB trees for (a) one, (b) two, (c) four, and (d) eight processors.

For maintaining virtual lists, we use the advanced global protocol for global scheduling and the standard protocol for other situations. In the global protocol, it appears that the degree of the global load balancer (GLB) tree should be moderately large because for a tree with small degree (say 2) we need more GLB processes each of which will incur some overhead. In [90], Sinha and Kale also implemented similar global scheduling by letting one load balancer handle at least 8 processors or 8 load balancers, i.e., they used a load balancer tree with a degree at least 8. But, in our actual implementation, since we will use at most 8 processors, we let the tree degree be 4 so that we can conduct experiments for a GLB tree with more than one level. Figure 6.2 shows GLB trees with $p = 1, 2, 4$, and 8, where the tree for 8 processors requires a two-level GLB tree and others require only a one-level GLB tree.

The GLB processes can be installed either on host processors or on CABs. Since most parallel systems do not use network coprocessors (such as CABs), it is natural for us to focus

on the case of installing GLB processes on host processors. However, we will also examine the case of installing them on CABs in order to see the benefit of using network coprocessors.

For the analysis of the experimental results in the following sections, we define *speedup* and *efficiency* in Definition 6.1 below. The time taken to run a job on one processor of the parallel version is generally higher than that for a sequential program because the parallel version also includes some scheduling, communication, and GLB overhead. Therefore, it makes more sense to use the timing results for the sequential version as a baseline.

Definition 6.1 *Let T_{seq} be the execution time for the original sequential version of a program. Also, let T_p be the time for the parallel version of the program on p processors. We define the* SPEEDUP *as $S_p = T_{seq}/T_p$ and the* EFFICIENCY *as $E_p = S_p/p$. If we use T_1 instead of T_{seq}, we call these measures* SIMPLE SPEEDUP, *$S'_p = T_1/T_p$, and* SIMPLE EFFICIENCY, *$E'_p = S'_p/p$.*

6.2 Fibonacci

In this section we will present a simple Fibonacci problem, which will allow us to derive the average overhead for one task and to investigate the case in which tasks have a minimum number of operations.

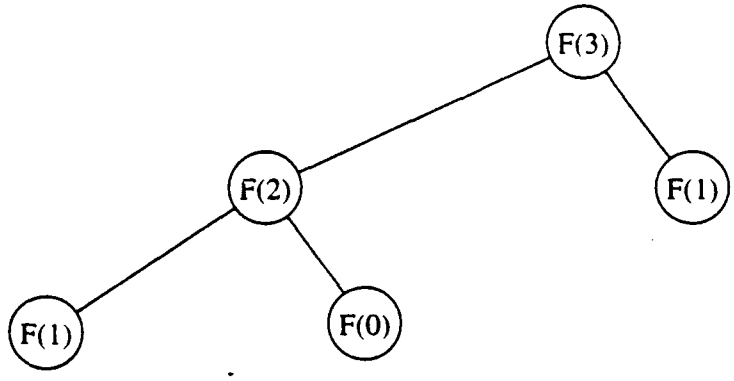


Figure 6.3: The computation tree for $F(3)$.

The Fibonacci problem solves the function $F(n)$ recursively as follows.

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

We can recursively expand this computation as a tree, as illustrated in Figure 6.3 with $n = 3$. This can be considered as a D&C algorithm. The program can be made parallel by letting each node in the computation tree represent a thread. After the node for $F(n)$ creates two children for $F(n-1)$ and $F(n-2)$, the node is suspended. The node becomes executable again only after receiving two returned values $F(n-1)$ and $F(n-2)$ from its two children.

We will use PDC as the scheduling algorithm for this parallel program. As described in Section 3.1.2, each node initially has local priority $\pi_L = l$ and global priority $\pi_G = -l$, where l is the level of the node in the tree. When the node resumes from suspension (i.e., when it receives two returned values from its two children), π_G of the node becomes a very low number, say $-\infty$. The reason is that since the grain size of the node at this moment becomes very small (see the grain size definition in Section 4.1.2.3), we prefer not to move the node to another processor.

# processors (p)	1	2	4	8
Total Time (T)	53.88	27.61	13.54	6.84
Simple Speedup (S')	1.00	1.95	3.97	7.86
Simple Efficiency (E')	1.00	0.98	0.99	0.98

Table 6.1: The total times (in seconds) and simple speedups/efficiencies for parallel Fibonacci.

In our experiment, we computed $F(26)$ with the GLB installed on a CAB so that we may ignore the GLB overhead. The performance results are shown in Table 6.1. Task creation and scheduling (i.e., task insertion and deletion) dominate most of the computation time because other operations require short times, for the following reasons. The essential computation for Fibonacci consists of only addition and assignment, which is insignificant when compared with the overhead of task scheduling. In addition, we also observe little communication, i.e., 98 sends/receives (about 20 milliseconds in total) for each processor. Since computing $F(26)$ requires scheduling nodes 589252 times (including the times for rescheduling nodes which resume from suspension), the average time for one task scheduling and task creation is about 91.4 microseconds (53.88 seconds /589252).

The results for simple speedup also show that the mechanism for task scheduling and creation can be parallelized well. For example, the simple speedup for 8 processors is as high as 7.86 and the simple efficiency is 98%.

6.3 Set Covering

Set covering is an important application of integer linear programming in the area of mathematical optimization. Balas and Padberg [9] surveyed many real applications of set covering in industry and listed over 10 types. Examples includes airline crew scheduling [22], truck delivery [10], facility location [83], switching circuit design [84], political districting [36], and information retrieval [30].

The *set covering* problem is

$$\min\{cx \mid Ax \geq e, x_j = 0 \text{ or } 1, \forall j, 1 \leq j \leq n\}$$

where A is an $m \times n$ matrix of zeros and ones, c is a row vector of n positive (integer) weights, and e is a column vector of m ones. The above definition can be interpreted as follows: Each column a_j of A is associated with a set M_j (associated with a weight c_j) with the following properties: (1) M_j is a subset of $M = \{1, \ldots, m\}$; (2) each value i is in M_j if and only if $a_{ij} = 1$. The set covering problem is to find a minimum-weight family of subsets M_j, $1 \leq j \leq n$, which covers all elements of M. For example, for airline crew scheduling, M corresponds to the set of flight legs (nonstop flights from one city to another) to be covered, while each subset M_j corresponds to a possible tour (starting and ending at the same point via a sequence of flight legs) for a crew. Let each tour be associated with a cost. The problem is then to obtain a minimum-cost collection of these tours to cover all the flight legs in M. According to the survey in [9], there are usually hundreds of possible tours and thousands of flight legs to be covered. In our experiments, we chose problems with 200 possible tours, and with 1000 flight legs to be covered (the same size was commonly used in [8, 35, 41]).

The set covering problem, as well as other mathematical optimization problems, usually requires large-scale branch-and-bound (B&B) tree search in order to locate the optimal solution. We have found that these searching processes are highly dynamic in the sense that it is difficult to make useful a priori estimates on the number of nodes the search tree will explore and the

sizes of the nodes. This indeterminacy, plus the possible large load fluctuations in the network, makes the problem of solving set covering over networks extremely difficult. Since both D&C and BFS can be used to implement B&B tree search, the set covering problem is a good application example to test both PBFS and PDC scheduling algorithms in our experiments.

The most advanced technique [8, 35, 41] for solving the set covering problem, as well as other mathematical optimization problems [73], is based on the following two steps:

1. Convert it to a linear programming (LP) problem and find the optimum LP solution, which may be a floating-point number. Usually, a LP problem can be solved very efficiently. For example, Karmarkar [55] proposed a polynomial time algorithm to solve the LP problem. In addition, Sethi and Thompson also designed a specialized *pivot and probe algorithm*, called *PAPA* [88], to solve the LP problem efficiently.

2. Apply a B&B technique to search for the desired optimal *integer* solution starting from the optimal LP solution. Since the LP solution is usually close to the integer solution, the search tree will be much smaller than that without using this technique of utilizing the LP solution.

The sequential set covering program for our experiments was originally written by Harche and Thompson [41]; it was optimized by Wu *et al.* [106]. This program takes the LP result generated by the PAPA package [88] and then performs B&B tree search[1] interactively. This program is described as follows:

1. The user chooses an upper bound and the tree search strategy, e.g., incomplete/complete tree search and depth-/best-first search. Incomplete tree search is allowed because in real applications getting a decent solution quickly is sometimes better than getting the optimal solution in an enormous length of time.

2. The program uses the chosen strategy to search for all the solutions with costs less than the given upper bound. The program will report the minimum cost of these solutions, if one exists.

[1] In the B&B tree search, only one column of data in the set covering matrix is transferred between any two nodes of the B&B tree.

Regarding parallelization of B&B programs, researchers [63, 66] have reported the *speedup anomaly phenomenon* in which the speedup is irregular (sometimes superlinear and sometimes sublinear) when different numbers of processors are used. Since the order of expanding nodes may vary when a different number of processors is used, the search may need to expand significantly more nodes or fewer nodes.

Since this anomalous phenomenon would make our performance analysis difficult, we choose cases in which the phenomenon does not occur. This ensures a rigorous standard in our testing. For our experiments, we choose a problem in which no solution will be found in the search; thus, we would prune no nodes at runtime. Thus, the expanded tree is always the same in spite of different search orders. Note that the tree shape is still very irregular.

Based on the above criterion, the experiments use four problems, called Problems 1, 2, 3, and 4, of increasing sizes. In these four problems, the heights of the search tree are 30, 36, 66, and 71, respectively; the numbers of tree nodes are 1135, 2107, 7097, and 23767, respectively.

In our experiments on the set covering problem, we apply both PBFS and PDC scheduling algorithms, described in Sections 6.3.1 and 6.3.2, respectively. In Section 6.3.3, we show the experimental results obtained by installing the GLB tree on CABs. In addition, we also apply a scheduling algorithm, called *parallel hybrid search* (*PHS*), to the set covering problem in Section 6.3.4. This shows how easily we can change to different task scheduling algorithms to possibly obtain better scheduling.

6.3.1 PDC

Table 6.2 lists the performance results for Problems 1 through 4 with the PDC scheduling algorithm. The results for speedups are also depicted in Figure 6.4. These results show that the speedup increases with the problem size. In particular, on eight processors, the largest problem (Problem 4) has a very good speedup of 7.71. This is because, in general, the number of nodes N in a tree grows exponentially with the tree height h. For example, for a complete binary tree, when h increases by one, N is doubled. Therefore, on the average, the larger a problem, the more independent tasks are available; so the better the efficiency.

In order to understand the performance results of Table 6.2, we will examine the overhead

	p	1(seq)	1	2	4	8
Problem 1	T	16.01	16.29	8.51	4.66	3.05
	S	1.00	0.98	1.88	3.44	5.25
	E	1.00	0.98	0.94	0.86	0.66
Problem 2	T	31.60	31.97	16.48	8.73	5.01
	S	1.00	0.98	1.92	3.62	6.31
	E	1.00	0.99	0.96	0.91	0.79
Problem 3	T	176.58	177.36	89.24	45.79	24.64
	S	1.00	1.00	1.98	3.86	7.17
	E	1.00	1.00	0.99	0.96	0.90
Problem 4	T	464.45	466.74	233.38	118.00	60.24
	S	1.00	1.00	1.99	3.94	7.71
	E	1.00	1.00	1.00	0.98	0.96

Table 6.2: Measured performance results for the PDC scheduling algorithm (T: Time in seconds, S: speedup, and E: efficiency).

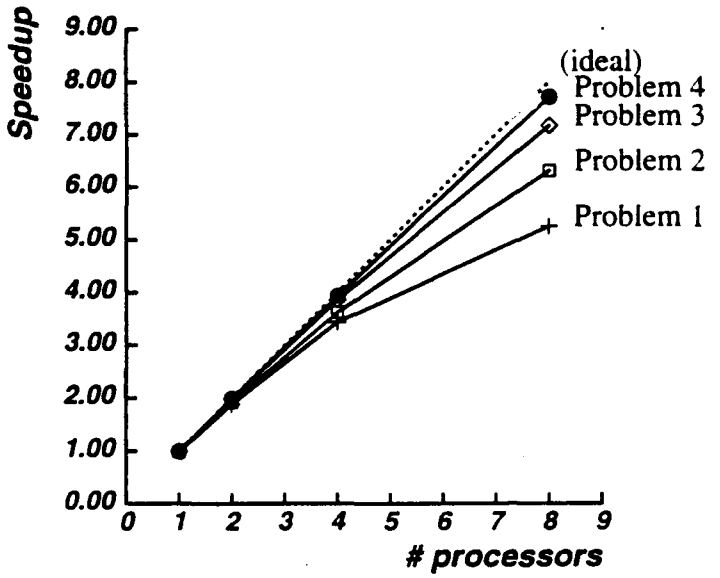


Figure 6.4: Speedups for Problems 1-4 with PDC.

	1	2	4	8
Problem 1	0.09	0.04	0.02	0.01
Problem 2	0.18	0.08	0.04	0.02

Table 6.3: Averaged scheduling overhead in seconds when using the PDC scheduling algorithm.

and the idle time for Problems 1 and 2 which are representative for our analysis. There are two main kinds of overhead: the overhead for scheduling each task locally (without any communication) and the overhead for communication. Table 6.3 shows the average scheduling overhead on each processor for Problems 1 and 2. The overhead for scheduling each task locally is about 80 microseconds (= 0.09 seconds /1135) from the table. Since the average task computation time is about 15 milliseconds (which is 16.01 seconds/1135 tasks), the scheduling overhead is less than 1%. Since we always search the same complete tree, the overhead for scheduling local tasks can be distributed over processors nearly evenly.

	height	1	2	4	8
Problem 1	30	15	188	405	1154
Problem 2	36	8	265	502	1366

Table 6.4: Number of sends/receive pairs for the PDC scheduling algorithm.

To analyze the communication overhead, we measured the number of sends/receives for Problems 1 and 2; results are shown in Table 6.4. Since each send/receive is roughly 200 microseconds [27], we can roughly calculate the aggregate communication overhead from the table. For PDC, the ratio of the number of sends/receives to the product of the tree height and the number of processors is close to a constant between 3 and 5. This result is close to the theoretical communication cost [105], which is $O(ph)$.

Recall the advanced global protocol in Section 4.1.2.3, which requires a second broadcast during each load balancing round if the total load in T'_i is small. In our experiments, we only need the second broadcast at most once for each run. This result is close to the theory in Section 4.1.2.3.

Processors become idle mainly in situations which lack parallelism or have long latency for responding to task requests. In our implementation, we use a scheme in which a processor schedules the next task while executing the current task. Since the task granularity is quite large when compared with the latency of task request (each task request requiring about 3 to 5 sends/receives), processors become idle mainly due to lack of parallelism. Note that if the task request latency is very large (e.g., when using a wide-area network), our system can schedule several tasks at a time so that the latency can be hidden.

Table 6.5 shows the idle times for Problems 1 and 2. In general, the more processors that are

	1	2	4	8
Problem 1	0.00	0.30	0.47	0.76
Problem 2	0.00	0.35	0.61	0.76

Table 6.5: Average idle times (in seconds) when using the PDC scheduling algorithm.

used, the more likely they are to become idle. For the trees in the set covering problem of our experiments, the average branching factor (defined as $N^{1/h}$) is very low, about 1.1 to 1.3. So, for example, when we initially execute the part of the tree near the root, not much parallelism can be exploited. When the number of processors increases, we expect the maximum value of the average idle time to be close to the execution time of all the nodes on the critical path from the root to the leaves. Since the number of nodes on the critical path (about $O(h)$) is small compared with the total number of nodes (N), the idle time for the PDC scheduling algorithm can be negligibly small.

6.3.2 PBFS

	p	1(seq)	1	2	4	8
Problem 1	T	16.79	17.71	9.42	5.16	3.52
	S	1.00	0.95	1.78	3.25	4.77
	E	1.00	0.95	0.89	0.81	0.60
Problem 2	T	33.48	34.83	17.85	9.44	5.68
	S	1.00	0.96	1.88	3.55	5.89
	E	1.00	0.96	0.94	0.89	0.74
Problem 3	T	181.32	185.62	94.87	48.31	25.83
	S	1.00	0.98	1.91	3.75	7.02
	E	1.00	0.98	0.96	0.94	0.88
Problem 4	T	475.31	486.84	247.03	125.77	64.81
	S	1.00	0.98	1.92	3.78	7.33
	E	1.00	0.98	0.96	0.95	0.92

Table 6.6: Measured performance results for the PBFS scheduling algorithm (T: Time in seconds, S: speedup, and E: efficiency).

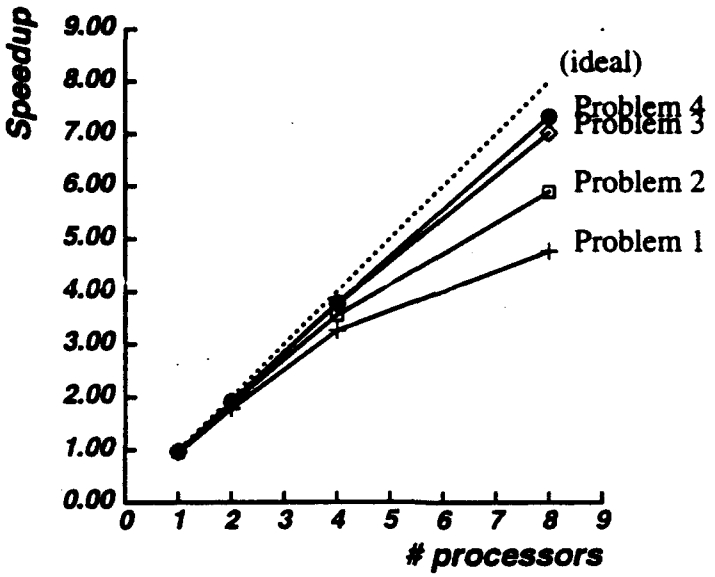


Figure 6.5: Speedups for Problems 1-4 with PBFS.

Table 6.6 lists the measured performance for Problems 1 through 4 for the PBFS scheduling algorithm. The results for speedups are also depicted in Figure 6.5. Note that the PBFS results are not as good as the PDC results. This is mainly because in Section 5.1, the PDC scheduling algorithm employs a provably minimum number of interprocessor communication messages.

	Nodes	1	2	4	8
Problem 1	1135	770	1190	1695	1846
Problem 2	2107	1672	1963	2716	3543

Table 6.7: Measured Number of sends/receives when using the PBFS scheduling algorithm.

For PBFS, the idle time and the scheduling overhead are very similar to those for PDC and their presence is for the same reasons discussed before. So, here we will only discuss the communication overhead for PBFS. Table 6.7 shows the measured numbers of send/receive pairs for Problems 1 and 2. The numbers for PBFS are much larger than those for PDC. This is because in PBFS whenever a task is scheduled, the minimum-cost task needs to be recomputed across all processors. As mentioned above, the communication cost for PBFS is $O(V)$. If we check each column, the ratios of the amount of communication to the number of nodes are nearly constant. But, when we use more processors, the ratios go up. This is because in a realistic situation we need more control packets to help send a cross node out when using more processors. However, our system only lets the ratio grow from 0.7 (for one processor) to 1.7 (for 8 processors).

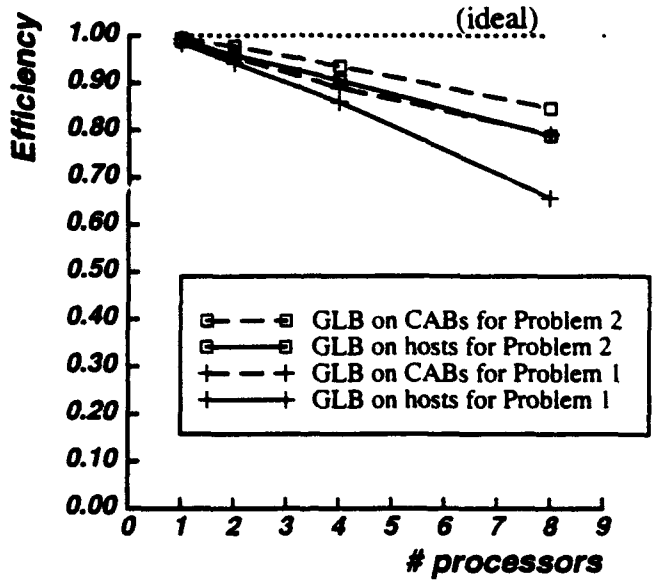


Figure 6.6: Efficiencies with PDC (installing GLB on CABs)

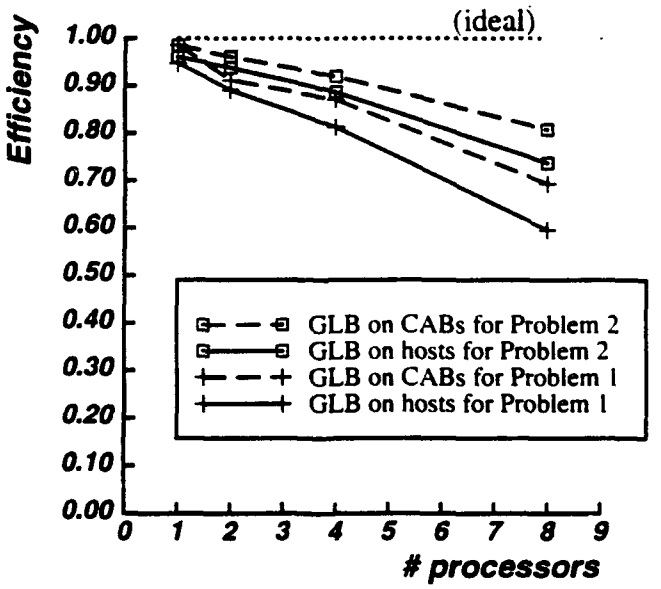


Figure 6.7: Efficiencies with PBFS (installing GLB on CABs)

6.3.3 Global Load Balancer on CABs

In this section, we will investigate the effect of installing the global load balancer (GLB) tree on CABs. When the GLB is installed on CABs, the GLB work is "off-loaded" from hosts to CABs and therefore applications should get better performance.

Figure 6.6 (6.7) shows efficiencies of the PDC (PBFS) scheduling algorithms for Problems 1 and 2. The dashed lines are results obtained when we install the GLB tree on CABs; the solid lines are the same results as those in Sections 6.3.1 and 6.3.2, which are obtained when we install the GLB tree on hosts.

The performance data indicate that we can improve the efficiency by about 2% to 20% for PDC and 5% to 20% for PBFS by installing the GLB on CABs. The improvement is small because the task grain size (about 15 millisecond) is large when compared with the time for each send/receive (about 200 microseconds). In general, installing GLB on CABs will help more for those applications requiring heavy communication, such as problems with small-grained tasks. Our results also show that we can save more time for PBFS (which requires more communication) than for PDC.

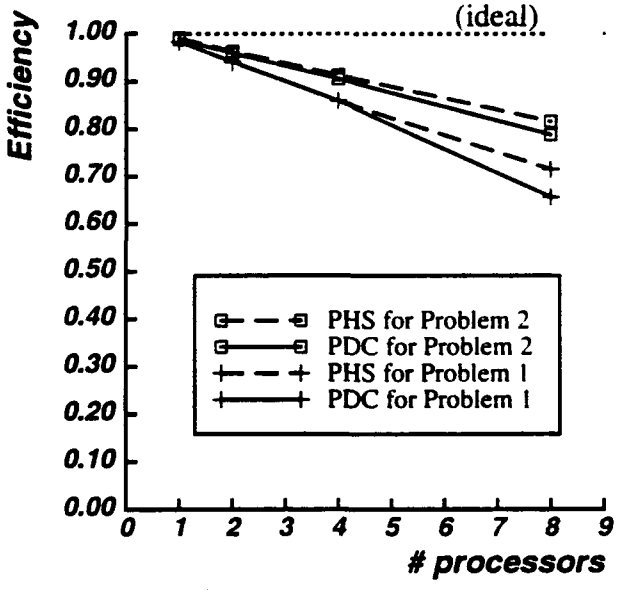


Figure 6.8: Performance results for parallel hybrid search.

6.3.4 Parallel Hybrid Search

In the set covering problem, the computation tree is usually lopsided. A node with a smaller cost usually has a bigger subtree. So, we can change PDC by assigning to the node $\pi_G = -c$, where c is the cost of the node. This is called *parallel hybrid search* (*PHS*) in the sense that it behaves like PDC locally (on each processor) while it behaves like PBFS globally (over the whole system).

The performance results for Problems 1 and 2 with PHS are shown in Figure 6.8. We observe that PHS is slightly better than PDC, by at most a 10% margin. Most importantly, this experiment demonstrates how easily we can change our scheduling algorithm to obtain possibly better results.

6.4 Summary

We summarize our experimental results as follows:

- The average overhead for scheduling/creating one local task can be as low as 80 microseconds. The scheduling overhead can be parallelized well too.
- For the set covering problem, we generally get good speedups for both PBFS and PDC. For a specific large problem requiring less than 500 seconds, we observed a speedup of 7.71 with PDC and 7.33 with PBFS on 8 processors.
- The number of sends/receives for PDC is about $O(ph)$ while that for PBFS is about $O(N)$. So, PDC can be scaled up well. For PBFS, a high-speed network can help improve performance significantly.
- For PDC and PBFS, the maximum value of the average idle time can be close to the execution time of all the nodes on the critical path from the root to leaves. Since the number of the nodes on the critical path (h) is small compared with the total number of nodes (N), the idle time for the two scheduling algorithms can be negligibly small.
- We can "off-load" the computation of the GLB to network coprocessors, if they exist.

- Although parallel hybrid search does not improve performance significantly, it serves to demonstrate how easily we can change our scheduling algorithm to obtain possibly better results. This point is especially important for parallel programmers who come up with many possible scheduling algorithms and want to compare them empirically.

Chapter 7

Conclusions

7.1 Summary

In this thesis, we have proposed a general parallel programming model, called the *multilist scheduling model*, which decomposes task scheduling into (1) the specification of scheduling policies and (2) the implementation of supportive scheduling operations (e.g., routines for maintaining task lists and handling interprocessor communication for load balancing), and then hides the latter details from the programmer. This model is based on a uniform scheduling model involving the use of multiple scheduling lists. The system has the following advantages:

1. *Ease of use.* Under this new model, programmers only need to specify scheduling policies based on scheduling lists in order to implement scheduling algorithms; they do not need to write the details of supportive scheduling routines. In fact, the supportive scheduling routines are the most difficult and time-consuming part to write. Typically they require thousands of lines of code in C. This was the case in our earlier experience [31, 62] in parallelizing Noodles, a solid modeling program [33] which took 3 months to write the load balancing part! In sharp contrast, the code for the *PDC-WK* and *PBFS-GPQ* scheduling algorithms (shown in Appendix A.2) only has about 10-20 lines. A program of this size can be written within tens of minutes.

2. *Generality.* We have shown that our new model results in no loss of generality with respect to the standard scheduling model (see Chapter 2). That is, we can recast any scheduling algorithm in terms of our multilist scheduling model. We have illustrated the generality of the model by implementing nine scheduling algorithms (in Chapter 3) based on the model — two for parallel divide-and-conquer and two for parallel best-first search, two for parallel network simulation, one for parallel quicksort, one for parallel loops, and one for parallel alpha-beta search.

3. *Efficiency.*

   - We have shown that our general approach incurs no significant performance overhead at least for the scheduling algorithms for parallel BFS and D&C. Although the ultimate goal would be to show that our general approach incurs no significant overhead for *any* scheduling algorithm, this goal appears to be impossible to meet. Our limited success, however, is still significant. In the past, it was unclear how to support scheduling algorithms for both parallel BFS and D&C in a uniform framework [34]. We believe our system is the first that can do so.

   - We have devised an efficient technique to cope with the main problem of task prioritization which arises when there is a sparse distribution of priorities. When few tasks have the same priority, it would be inefficient to simply perform load balancing by considering only those tasks that have the highest priority. Our approach is to use a novel algorithm, called the parallel range selection (PRS) algorithm (see Section 5.2), to try to select additional highest-priority tasks, such that the total computation time of these tasks is comparable to the maximum overhead for load balancing. Then, we schedule tasks from this set in order to balance the load. In Section 5.2, we proved that the PRS algorithm only requires one combining and disseminating operation (defined in Section 4.1.2.2), and each packet size in the algorithm is only $O(\log^2 p)$, where p is the number of processors. This results show that the PRS algorithm performs very efficiently on network-based multicomputers.

   - We have obtained good experimental results. For example, for a specific set covering problem requiring less than 500 seconds, we obtain a speedup of 7.71 for parallel D&C and 7.33 for parallel BFS on Nectar with 8 processors.

7.2 Contributions

The main contribution of this thesis is a new approach for parallel task scheduling, *multilist scheduling*, which is easy to use, general, and efficient. This new approach successfully decomposes the task scheduling process into the specification of scheduling policies and the supportive scheduling routines such that the scheduling programmers can focus on designing efficient scheduling algorithms, while the system designers can focus on designing efficient supportive routines.

While proposing this new model, we also make the following contributions:

- We develop some multilist scheduling schemes to implement nine scheduling algorithms — two for parallel divide-and-conquer and two for parallel best-first search, two for parallel network simulation, one for parallel quicksort, one for parallel loops, and one for parallel alpha-beta search.

- We show that the multilist scheduling model results in no loss of generality with respect to the standard scheduling model (see Chapter 2).

- We present an efficient scheduling algorithm for parallel D&C and prove that, among all the scheduling algorithms which can split the load nearly evenly, our algorithm is optimal with respect to the communication cost.

- We design an efficient PRS algorithm for the situation of sparse priority distribution and prove that the algorithm only needs one combining and then one disseminating operation and each packet size is only $O(\log^2 p)$, where p is the number of processors.

- We design an efficient data structure for the operations, INSERT, DELETE, MAXPRI, DELETEMAX, THRESHPRI, and SPLIT (see Section 4.2). We also prove that the computation times for all the above operations are $O(\log n)$, where n is the number of distinct priorities in the priority queues, and the amortized times for the above operations which access/insert/delete a task with the highest or lowest priority are $O(1)$.

- We demonstrate good performance results for the scheduling algorithms for parallel BFS and D&C.

Multilist scheduling is the first approach which can hide the details of supportive scheduling routines while simultaneously supporting general task scheduling. We expect that this thesis will have a significant impact on future parallel programming, especially in the domain of multicomputers.

7.3 Future Work

The above research can be extended in the following directions.

- Apply the model to more applications, such as operations research problems (e.g., the traveling salesman problem [76]), alpha-beta search problems, network simulation problems, scientific problems, and any other interesting problems.
- Formalize a language/interface. This also includes two important parallel programming issues: handling global variables [14, 15] and calling remote procedures [7, 13, 54, 93, 96, 102] over distributed-memory systems.
- Add fault tolerance to the system. It is especially important for a network to tolerate processor failure. For example, a remote workstation may be turned off unexpectedly.
- Develop tools on top of this package for debugging, performance monitoring [19], and graphical development [12].
- Port our model to other parallel systems, e.g., CM5 [99] and iWarp [18]; also port the programming system to other network interfaces, e.g., PVM and sockets of TCP/IP.

Appendix A

User Interface

The current multilist scheduling system is operational at CMU. It has a C language interface. This interface is presented in Section A.1 in enough detail to illuminate the code of the *PBFS-GPQ* and *PDC-WK* scheduling algorithms, which will be presented in Section A.2.

A.1 Interface Definitions

To distinguish our interface from system calls or variables, we prefix each function name with "`MLS_`" (MultiList Scheduling).

A.1.1 Initializing the Multilist Scheduling System

```
void MLS_Init (spec, pkey_type, min_grain);
```

The `MLS_Init()` procedure initializes the multilist scheduling system.

The parameter `spec` is the name of a user-defined routine which specifies the scheduling pattern, i.e., the declaration of PLs and the merge patterns for VLs (all of which will be defined later). In the current implementation, the routine can only be executed inside `MLS_Init()`.

The parameter `min_grain` is an integer, representing the minimum task grain size used in the computation. The system currently assumes that all tasks have the same grain size. This parameter provides the task grain size so that the system can easily set up some system parameters for load balancing. In our current system, since the overhead for task scheduling and creation is already 80 microseconds (as shown in Section 6.3) on a Sun4/330, we let one unit of grain size stand for 100 microseconds on Sun4/330s.

The parameter `pkey_type` determines the type of priority keys used in the system. The choices are integer, string, bit string, etc. But, since the integer type is most common, we only support the integer type (`MLS_INT_KEY`) in the current implementation. The work in [85] uses bit string as the key type.

Since our system is in the SPMD (single program, multiple data) style [56], the program is replicated on each processor. Each instance of the program (on a distinct processor) must explicitly call `MLS_Init()` before calling any other functions which are described below for the multilist scheduling system.

A.1.2 Physical Lists

There are two kinds of physical lists (PLs): base PLs and derived PLs.

```
MLS_List_p MLS_Base_PL (list_name, list_type,
                        lb, ub, Indiv_Range);
```

The `MLS_Base_PL()` procedure creates a base PL with the name `list_name`, the type `list_type`, the expected priority range between `lb` and `ub` (inclusive) and the indivisible function `Indiv_Range`, and then returns a pointer to the PL with type `MLS_List_p`. Note that the word `list` in our interface, by convention, represents PL.

The parameters `lb` and `ub` respectively specify the lower bound and upper bound of the priority range in this PL. If the bounds are not known *a priori*, the user can use the complete priority range, whose lower bound and upper bound are respectively `MLS_MIN_PRI` and `MLS_MAX_PRI`, defined by the system.

The parameter `Indiv_Range` is some user-defined function `IR(pri)`: given `pri`, the function `IR()` returns a priority range containing the priority `pri`. The PL can use this function to determine if a new maximum priority is out of the original priority range. If so, the system may need to report to another processor. (See Section 4.1.1.) If the parameter `Indiv_Range` is null, this implies that each priority is a distinct indivisible priority range. If the PL will not be merged into any VL on another processor, the parameter `Indiv_Range` has no effect.

We currently restrict the list type to 2-4 trees (see Section 4.2.1) to implement PLs. But, in the future, the parameter `list_type` may be used to specify another type of list, in case we discover some type of list which is more efficient for some cases.

If this procedure is called for the jth time on processor P_i, the created PL is designated PL_{ij}. In the current version, we simplify operations by assuming that all jth lists PL_{*j} have the same list name and have the same parameters. So, we do not need communication to set up associated links (in the scheduling pattern).

```
MLS_List_p MLS_Derived_PL (list_name, base, func, inv
                           lb, ub, Indiv_Range);
```

The `MLS_Derived_PL()` procedure creates a derived PL which is based on a base PL `base` with the priority translation function `func`, whose inverse function is `inv`. The parameters `list_name`, `lb`, `ub`, and `Indiv_Range` are the same as those for base PLs. If this PL is not part of a global scheduling subpattern, the inverse function `inv` will not be used. Note that this operation for derived PLs has not yet been implemented.

A.1.3 Merging Physical Lists into Virtual Lists

In the current implementation, we provide the following functions to specify how to merge PLs into VLs:

```
void MLS_Merge (list_name, procx);
void MLS_Merge_Local (list_name);
```

Assume that the procedure `MLS_Merge()` is executed on a processor P_i. Then, this procedure specifies that a PL named `list_name` on processor P_j will be merged into VL_i according to the standard protocol, where $j =$ `procx`. If $j = i$, we can use `MLS_Merge_Local` instead. In any case, a system variable, called `MLS_this_proc` ($= i$), is provided to refer to the current processor P_i.

```
void MLS_Merge_All(list_name);
```

Assume that the procedure `MLS_Merge_All()` is executed on a processor P_i. Then, this procedure specifies that all the PLs named `list_name` (over the whole system) will be merged into the VL on processor P_i. If every processor calls the same procedure, the system can apply the advanced global protocol to these PLs.

```
void MLS_Merge_Dynamic(list_name, pset);
```

Assume that the procedure `MLS_Merge_Dynamic()` is executed on a processor P_i. This procedure specifies that all the PLs named `list_name` in a set S of processors will be merged into the VL on processor P_i, but the processor set S can be dynamically determined as follows: Whenever processor P_i tries to schedule a task and needs to check the PLs named `list_name`, processor P_i will execute the function `pset()` again to obtain the new set S, and then find the highest priority among PLs on the processors in S.

Let us consider the example of *PDC-RR* scheduling algorithm. The programmer can call this function, as follows.

```
MLS_Merge_Dynamic("GL", rr_pset);
```

The function `rr_pset()`, provided by the programmer, returns the next processor in a round-robin fashion. More specifically, when called, this function lets the variable $s = (s \bmod p) + 1$ and returns s so that the system knows that the processor set includes processor P_s. Since the programmer also needs to specify the priority ranges of GL and LL such that the system knows that each priority in LL is no less than that in GLs, the processor calls the function and schedules a task from a GL only when there are no local tasks. Thus, the system will perform as described in [34, 44, 82].

A.1.4 Priority Assignment

```
void MLS_Task_Pri (PL, T, pri);
void MLS_Change_Pri (PL, T, pri);
```

The procedure `MLS_Task_Pri()`, assumed to be called on processor P_i, inserts task `T` into the PL `PL` on processor P_i according to the priority `pri`. But, if the value of `pri` is `MLS_UNDEF_PRI`, the system will not insert the task into the PL. Note that the current implementation does not support the feature of creating a task on another processor.

Similarly, the `MLS_Change_Pri()` procedure, assumed to be called on processor P_i, changes the priority of task `T` to `pri`.

A.2 Examples

Using the above interface definitions, we will illustrate the code for the *PDC-WK* and *PBFS-GPQ* scheduling algorithms, below.

```
/*** following are in the file PBFS.h ***/
extern void PBFS_Schd();    /* scheduler for PBFS */
MLS_List_p  LT;             /* scheduling list */

/* interface for the programmer writing BFS application code. */
#define DECLARE_NODE(T,c) MLS_Task_Pri(LT,T,-c)
#define INIT_PBFS(g) MLS_Init(PBFS_Schd, g, INTKEY)

/*** following are in the file PBFS.c ***/
void PBFS_Schd () {
    /* create a PL */
    LT = MLS_Base_PL ("LT", MLS_INT_KEY, MLS_MIN_PRI, MLS_MAX_PRI, NULL);

    /* merge all PLs (named "LT") in the entire system */
    MLS_Merge_All("LT");
}
```

Figure A.1: The code for the *PBFS-GPQ* scheduling algorithm.

```
/*** following are in the file PDC.h ***/
extern void PDC_Schd();    /* scheduler for PDC */
MLS_List_p  LL, GL;        /* local and global scheduling lists */

/* interface for the programmer writing D&C application code. */
#define DECLARE_NODE(T,l) \
    { MLS_Task_Pri(T,LL,l); MLS_Task_Pri(T,GL,-l); }
#define INIT_PDC() MLS_Init(PDC_Schd, DC_GRAIN, INTKEY)

/*** following are in the file PDC.c ***/
void PDC_Schd () {
    /* create two PLs */
    LL = MLS_Base_PL ("LL", MLS_INT_KEY, 0, MLS_MAX_PRI, NULL);
    GL = MLS_Base_PL ("GL", MLS_INT_KEY, MLS_MIN_PRI, 0, NULL);

    /* merge the local LL and all GLs */
    MLS_Merge_Local("LL");
    MLS_Merge_All("GL");
}
```

Figure A.2: The code for the *PDC-WK* scheduling algorithm.

Bibliography

[1] Abdelrahman, T. and Mudge, T.
Parallel B&B Algorithms on Hypercube Multiprocessors.
in: **Proc. 3th Conf. on Hypercube Concurrent Computers and Applications.**
1988, pp. 1492–1499.

[2] Adam, T., Chandy, K., and Dickson, J.
A Comparison of List Schedules for Parallel Processing Systems.
Communications of the ACM, vol. 17 (1974), pp. 685–690.

[3] Aho, A., Hopcroft, J., and Ullman, J.
The Design and Analysis of Computer Algorithms.
Addison and Wesley, 1974.

[4] Akl, S., Barnard, D., and Doran, R.
Design, Analysis and Implementation of a Parallel Tree Search Algorithm.
IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 4 (1982), pp. 192–203.

[5] Anderson, S. and Chen, M.
Parallel Branch-and-Bound Algorithms on the Hypercube.
in: **Hypercube Multiprocessors**, edited by M. Heath.
SIAM Press, Philadelphia, 1987.

[6] Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., and Steenkiste, P. A.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
in: **Third Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III).**
Boston, Massachusetts, 1989.

[7] Bal, H.
The Shared Data-Object Model as a Paradigm for Programming Distributed Systems.
Vrije Universiteit, Amsterdam, Netherlands, 1989.

[8] Balas, E. and Ho, A.
Set Covering Algorithms Using Cutting Planes, Heuristics, and Subgradient Optimization: a Computational Study.
Mathematical Programming, vol. 12 (1980), pp. 37–60.

[9] Balas, E. and Padberg, M.

Set Partitioning: A Survey.
SIAM Review, vol. 18 (1976), pp. 710–760.

[10] Balinski, M. and Quandt, R.
On an Integer Program for a Delivery Problem.
Operations Research, vol. 12 (1964), pp. 300–304.

[11] Baudet, G.
The Design and Analysis of Algorithms for Asynchronous Multiprocessors.
School of Computer Science, Carnegie-Mellon University, 1978.

[12] Beguelin, A., Dongarra, J., Geist, G., Manchek, R., and Sunderam, V.
Graphical Development Tools for Network-Based Concurrent Supercomputing.
in: **Proceedings of Supercomputing '91**, IEEE.
Albequerque, 1991, pp. 435–444.

[13] Bershad, B., Ching, D., Lazowska, E., Sanislo, J., and Schwartz, M.
A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems.
IEEE Transaction on Software Engineering, vol. 13 (1987), pp. 880–894.

[14] Birman, K. and Joseph, T.
Exploiting Replication in Distributed Systems.
in: **Distributed Systems**, edited by S. Mullender.
Addison-Wesley, 1989, pp. 319–368.

[15] Birman, K., Joseph, T., Kane, K., and Schmuch, F.
ISIS – A Distributed Programming Environment User's Guide and Reference Manual.
Department of Computer Science, Cornell University, March 1988.

[16] Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., and Zagha, M.
Implementation of a Portable Nested Data-Parallel Language.
in: **Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.**
1993.

[17] Blum, M., Floyd, R., Pratt, V., Rivest, R., and Tarjan, R.
Time Bounds for Selection.
Journal of Computer and System Sciences, vol. 7 (1973), pp. 448–461.

[18] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J., and Webb, J.
iWarp: An Integrated Solution to High-Speed Parallel Computing.
in: **Proceedings of Supercomputing '88**, IEEE Computer Society and ACM SIGARCH.
Orlando, Florida, 1988, pp. 330–339.

[19] Bruegge, B.
BEE: A Basis for Distributed Event Environments (Reference Manual).
no. CMU-CS-90-180, Carnegie-Mellon University, November 1990.

[20] Bryant, R., Beatty, D., Brace, K., Cho, K., and Sheffler, T.
COSMOS: A Compiled Simulator for MOS Circuits.
in: **Proceedings of the Design Automation Conference**, ACM/IEEE.
1987, pp. 9–16.

[21] Campbell, M.
Algorithms for the Parallel Search of Game Trees.
no. Technique Report 81-8, Computer Science Department, University of Alberta, Canada, August 1981.

[22] Charnes, A. and Miller, M.
A Model for the Optimal Programming of Railway Freight Train Movements.
Management Science, vol. 3 (1956), pp. 74–92.

[23] Choi, Y.
Vertex-Based Boundary Representation of Non-Manifold Geometric Models.
Engineering Design Research Center, Carnegie Mellon University, 1989.

[24] Chrisochoides, N., Droegemeier, K., Fox, G., Mills, K., and Xue, M.
A Methodology for Developing High Performance Computing Models: Storm-Scale Weather Prediction.
in: **High Performance Computing Symposium 1993 Grand Challenges in Computer Simulation.**
1993, pp. 82–89.

[25] Chrisochoides, N., Houstis, E., and Rice, J.
Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers.
submitted to the special issue of the *Journal of Parallel and Distributed Computing* on Data-Parallel Algorithms and Programming, 1993.

[26] Cohn, R.
Source-Level Debugging of Automatically Parallelized Programs.
School of Computer Science, Carnegie-Mellon University, October 1992.

[27] Cooper, E., Steenkiste, P., Sansom, R., and Zill, B.
Protocol Implementation on the Nectar Communication Processor.
in: **Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols**, ACM.
Philadelphia, 1990, pp. 135–143.

[28] Cormen, T., Leiserson, C., and Rivest, R.
Introduction to Algorithms.
MIT Press, Cambridge, MA, 1989.

[29] Dantzig, G., Fulkerson, D., and Johnson, S.
Solution of a Large-Scale Traveling Salesman Problem.
Operations Research, vol. 2 (1954), p. 393.

[30] Day, R.
On Optimal Extracting from a Multiple File Data Storage System: An Application of Integer Programming.

Operations Research, vol. 13 (1965), pp. 482–494.

[31] Fahringer, T., Gubitoso, M., Kung, H., Prinz, F., and Wu, I.-C.
Parallelizing Noodles on Nectar.
internal document, CMU, 1989.

[32] Feldmann, R., Monien, B., Mysliwietz, P., and Vornberger, O.
Distributed Game Tree Search.
in: **Parallel Algorithms for Machine Intelligence**, edited by V. Kumar, P. Gopalakrishnan, and L. Kanal.
Springer-Verlag, 1990, pp. 66–101.

[33] Ferguson, C. and Korf, R.
Distributed Tree Search and its Application to Alpha-Beta Pruning.
in: **Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 1988)**.
Saint Paul, 1988, pp. 128–132.

[34] Finkel, R. and Manber, U.
DIB – a Distributed Implementation of Backtracking.
ACM Transactions on Programming Languages and Systems, vol. 9 (1987), pp. 235–256.

[35] Fisher, M. and Kedia, P.
Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics.
Management Science, vol. 36 (1990), pp. 674–688.

[36] Garfinkel, R. and Nemhauser, G.
Optimal Political Districting by Implicit Enumeration.
Management Science, vol. 16 (1970), pp. 495–508.

[37] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V.
PVM 3 User's Guide and Reference Manual.
no. ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[38] Guibas, L., McCreight, E., Plass, M., and Roberts, J.
A New Representation for Linear Lists.
in: **Proceedings of the Ninth Annual ACM Symposium on Theory of Computing**.
1977, pp. 49–60.

[39] Hamey, L., Webb, J., and Wu, I.-C.
An Architecture Independent Programming Language for Low-Level Vision.
Computer Vision, Graphics, and Image Processing, vol. 48 (1989), pp. 246–64.

[40] Haralick, R. and Elliott, G.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
Artificial intelligence, vol. 14 (1980), pp. 263–313.

[41] Harche, F. and Thompson, G.
The Column Subtraction Algorithm: An Exact Method for Solving Weighted Set Covering, Packing and Partitioning Problems.
To appear in **Computers and Operations Research**, 1993.

[42] Hoare, C.
Algorithm 63 (Partition) and Algorithm 65 (Find).
Communications of the ACM, vol. 4 (1961), pp. 321–322.

[43] Hoare, C.
Quicksort.
Computer Journal, vol. 5 (1962), pp. 10–15.

[44] Holliman, N.
Visualizing Solid Models: An Exercise in Parallel Programming.
Leeds University, September 1990.

[45] Hsu, F.-H.
Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.
School of Computer Science, Carnegie-Mellon University, 1990.

[46] Huang, S.-R. and Davis, L. S.
Parallel Interactive A Search: An Admissible Distributed Heuristic Search Algorithm.*
in: **11th International Joint Conference on Artificial Intelligence**.
Detroit, 1989, pp. 23–29.

[47] Huddleston, S. and Mehlhorn, K.
A New Data Structure for Representing Sorted List.
Acta Informatica, vol. 17 (1982), pp. 157–184.

[48] Hummel, S., Schonberg, E., and Flynn, L.
Factoring: A Method for Scheduling Parallel Loops.
Communications of the ACM, vol. 35 (1992), pp. 90–101.

[49] Ikudome, K., Fox, G., Kolawa, A., and Flower, J.
An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers.
in: **Proceedings of the Fifth Distributed Memory Computing Conference**, IEEE.
1990, pp. 1105–1114.

[50] *iPSC/2 C Programmer's Reference Manual.*
Intel, 1988.

[51] Jayasimha, D.
Communication and Synchronization in Parallel Computation.
Department of Computer Science, University of Illinois at Urbana-Champaign, September 1988.

[52] Jayasimha, D. and Loui, M.
The Communication Complexity of Parallel Algorithms.
no. CSRD 629, University of Illinois at Urbana-Champaign, 1986.

[53] Johnson, C.
Numerical Solutions of Partial Differential Equations by the Finite Element Method.
Cambridge University Press, 1987.

[54] Jones, M., Rashid, R., and Thompson, M.
Matchmaker: An Interface Specification Language for Distributed Processing.
in: **Conference Record of the 12th Annual ACM Conference on Principles of Programming Languages.**
1985, pp. 225–235.

[55] Karmarkar, N.
A New Polynomial-Time Algorithm for Linear Programming.
Combinatorica, vol. 4 (1984), pp. 373–395.

[56] Karp, A.
Programming for Parallelism.
IEEE Computer, vol. 20 (1987), pp. 43–57.

[57] Karp, R. and Zhang, Y.
A Randomized Parallel Branch-and-Bound Procedure.
in: **Proceedings of the 20th Annual ACM Symposium on Theory of Computing.**
Chicago, IL, 1988, pp. 290–300.

[58] Knuth, D. E. and Moore, R. W.
An Analysis of Alpha-Beta Pruning.
Artificial Intelligence, vol. 6 (1975), pp. 293–326.

[59] Kohler, W.
A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems.
IEEE Transaction on Computers, vol. 24 (1975), pp. 1235–1238.

[60] Kumar, V., Ramesh, K., and Rao, V. N.
Parallel Best-First Search of State-Space Graphs: A Summary of Results.
in: **Proceedings of the 1988 National Conference on Artificial Intelligence.**
1988, pp. 122–127.

[61] Kung, H.
Heterogeneous Multicomputers.
in: **Carnegie Mellon Computer Science: A 25-Year Commemorative**, edited by R. F. Rashid.
Addison-Wesley, 1990, pp. 235–251.

[62] Kung, H., Steenkiste, P., Gubitoso, M., and Khaira, M.
Parallelizing a New Class of Large Applications over High-Speed Networks.
in: **Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**, ACM.
1991, pp. 167–177.

[63] Lai, T. and Sahni, S.
Anomalies in Parallel Branch-and-Bound Algorithms.
Communications of the ACM, 1984, pp. 594–602.

[64] Lee, K.-F. and Mahajan, S.
Bill: a Table-Based, Knowledge-Intensive Othello Program.

no. CMU-CS-86-141, School of Computer Science, Carnegie-Mellon University, April 1986.

[65] Leffler, S., McKusick, M., Karels, M., and Quarterman, J.
The Design and Implementation of the 4.3BSD UNIX Operating System.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[66] Li, G.-J. and Wah, B.
Coping with Anomalies in Parallel Branch-and-Bound Algorithms.
IEEE Transaction on Computers, June 1986, pp. 568–573.

[67] Lin, F. and Keller, R.
The Gradient Model Load Balancing Method.
IEEE Transaction on Software Engineering, vol. SE-13 (1987), pp. 32–38.

[68] Maier, D. and Salveter, C.
Hysterical B-trees.
Information Processing Letters, vol. 12 (1981), pp. 199–202.

[69] Marsland, T. and Campbell, M.
Parallel Search of Strongly Ordered Game trees.
Computing Surveys, vol. 14 (1982), pp. 533–551.

[70] Nishikawa, H.
A Practical Load Balancing Scheme Embedded in Aroma.
private manuscript, 1992.

[71] Nishikawa, H. and Steenkiste, P.
Aroma: Language Support for Distributed Objects.
in: **International Parallel Processing Symposium.**
Los Angeles, 1992, pp. 686–690.

[72] Ortega, J. and Voigt, R.
Solution of Partial Differential Equations on Vector and Parallel Computers.
SIAM Review, vol. 27 (1985), pp. 149–240.

[73] Papadimitriou, C. and Steiglitz, K.
Combinatorial Optimization: Algorithms and Complexity.
Englewood Cliffs, NJ, 1982.

[74] Papadimitriou, C. and Ullman, J.
A Communication-Time Tradeoff.
SIAM Journal of Computing, vol. 16 (1987), pp. 639–646.

[75] Pearl, J.
Heuristics: Intelligent Search Strategies for Computer Problem Solving.
Addison-Wesley, 1984.

[76] Pekny, J. and Miller, D.
A Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems.
no. EDRC 05-27-88, Engineering Design Research Center, Carnegie Mellon University, May 1988.

[77] Plaxton, C.
Efficient Computation on Sparse Interconnection Networks.
Dept of CS, Stanford University, September 1989.

[78] Polychronopoulos, C. and Kuck, D.
Guided Self-Scheduling: A Practical Scheduling Scheme for parallel computers.
IEEE Transaction on Computers, vol. 36 (1987), pp. 1425–1439.

[79] Preparata, F. and Shamos, M.
Computational Geometry: an Introduction.
Springer-Verlag, New York, 1985.

[80] Printz, H.
Automatic Mapping of Large Signal Processing Systems to a Parallel Machine.
School of Computer Science, Carnegie-Mellon University, May 1991.

[81] Quinn, M.
Implementing Best-First Branch-and-Bound Algorithms on Hypercube Multicomputers.
in: **Hypercube Multiprocessors**, edited by M. Heath.
SIAM Press, Philadelphia, 1987.

[82] Rao, V. N. and Kumar, V.
Parallel Depth-First Search, Part I: Implementation.
International Journal of Parallel Programming, vol. 16 (1987), pp. 479–499.

[83] Revelle, C., Marks, D., and Liebman, J.
An Analysis of Private and Public Sector Location Models.
Management Science, vol. 16 (1970), pp. 692–707.

[84] Root, J.
An Application of Symbolic Logic to Selection Problem.
Operations Research, vol. 12 (1964), pp. 519–526.

[85] Saletore, V. and Kale, L.
Consistent Linear Speedups to a First Solution in Parallel State-Space Search.
in: **Proceedings of the 8th National Conference on Artificial Intelligence (AAAI 1990)**.
Boston, 1990, pp. 227–233.

[86] Samet, H.
Applications of Spatial Data Structures: Computer Graphics, Image processing, and GIS.
Addison-Wesley, Reading, MA., 1990.

[87] Schaeffer, J.
Distributed Game-Tree Searching.
Journal of Parallel and Distributed Computing, vol. 6 (1989), pp. 90–114.

[88] Sethi, A. and Thompson, G.
The Pivot and Probe Algorithm for Solving a Linear Program.
Mathematical Programming, vol. 29 (1984), pp. 219–233.

[89] Shu, W. and Kale, L. V.
A Dynamic Scheduling Strategy for Chare-Kernel System.
in: **Proceedings of Supercomputing '89**.
New York, NY, 1989, pp. 389–398.

[90] Sinha, A. and Kale, L.
A Load Balancing Strategy for Prioritized Execution of Tasks.
To appear in *International Parallel Processing Symposium*, 1993.

[91] Slate, D. and Atkin, L. R.
Chess 4.5 – The Northwestern University Chess Program.
in: **Chess Skill in Man and Machine.**
Springer-Verlag, 1977, pp. 82–118.

[92] Sleator, D. and Tarjan, R.
Self-Adjusting Binary Search Trees.
Journal of the ACM, vol. 32 (1985), pp. 652–686.

[93] Sollins, K.
Copying Structured Objects in a Distributed System.
Computer Networks, vol. 5 (1981), pp. 351–359.

[94] Steenkiste, P.
Nectarine - A Nectar Interface.
internal document, 1990.

[95] Steenkiste, P.
A Symmetrical Communication Interface for Distributed-Memory Computers.
in: **Proceedings of the Sixth Distributed Memory Computing Conference**, IEEE.
Portland, 1991, pp. 262–265.

[96] Sun Microsystems Inc.
Remote Procedure Call Protocol Specification.
Sun Microsystems Inc., February 1986.

[97] Tarjan, R.
Sequential Access in Splay Trees Takes Linear Time.
Combinatorica, vol. 5 (1985), pp. 367–378.

[98] Tarjan, R. and Van Wyk, C.
An O(n log log n)-Time Algorithm for Triangulating a Simple Polygon.
SIAM Journal of Computing, vol. 17 (1988), pp. 143–178.

[99] Thinking Machines Corporation.
The Connection Machine CM-5 Technical Summary.
Thinking Machines Corporation, January 1992.

[100] Tseng, P. S.
A Parallelizing Compiler for Distributed Memory Parallel Compiler.
Carnegie-Mellon University, May 1989.

[101] Wah, B. W. and Ma, Y.

Manip - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems.
IEEE Transaction on Computers, vol. 33 (1984), pp. 377–390.

[102] Weihl, W.
Remote Procedure Call.
in: **Distributed Systems**, edited by S. Mullender.
Addison-Wesley, 1989, pp. 65–86.

[103] Wu, I.-C.
Dual-Priority Task Scheduling: A New Parallel Programming Model for Tree Search Problems.
private manuscript (thesis proposal at CMU), 1991.

[104] Wu, I.-C.
Efficient Parallel Divide-and-Conquer for a Class of Interconnection Topologies.
in: **the Second Annual International Symposium on Algorithms**.
Taipei, 1991.

[105] Wu, I.-C. and Kung, H.
Communication Complexity for Parallel Divide-and-Conquer.
in: **1991 Symposium on Foundations of Computer Science**.
San Juan, 1991, pp. 151–162.

[106] Wu, I.-C., Thompson, G., and Harche, F.
private communication, 1993.

[107] Wu, M.-Y. and Shu, W.
Scatter Scheduling for Problems with Unpredictable Structures.
in: **Proceedings of the Sixth Distributed Memory Computing Conference**.
Portland, 1991, pp. 137–143.

[108] Zhang, Y.
Parallel Algorithms for Combinatorial Search Problems.
U.C. Berkeley, November 1989.